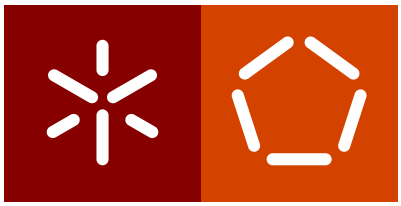
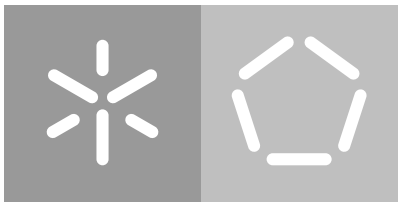


Universidade do Minho
Escola de Engenharia
Departamento de Informática

Daniel Carvalho da Cruz

Processamento Analítico Seguro

Outubro de 2018



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Daniel Carvalho da Cruz

Processamento Analítico Seguro

Dissertação de Mestrado
Mestrado Integrado em Engenharia Informática

Dissertação realizada sob a orientação de
Doutor João Tiago Medeiros Paulo
Professor Rui Carlos Mendes Oliveira

Outubro de 2018

AGRADECIMENTOS

Gostaria de agradecer a ajuda de várias pessoas, sem as quais este dissertação não seria possível.

Em primeiro, gostaria de agradecer ao meu orientador, Doutor João Paulo, e ao Ricardo Macedo por me guiarem e por toda a motivação que me deram ao longo deste projeto. Fico também agradecido pela disponibilidade e pelas inúmeras discussões que me permitiram melhorar e finalizar esta dissertação.

Agradeço também ao meu coorientador, Professor Rui Carlos Oliveira, pela oportunidade que me deu em me juntar ao grupo de investigação *HASLab*.

Aos colegas do grupo de investigação do *HASLab* pelo acolhimento e pelo bom ambiente de trabalho.

E, por fim, à minha família e amigos pelo apoio prestado ao longo do meu percurso académico.

ABSTRACT

Nowadays, users resort to multiple online applications and services to improve their lives, leading to the generation and processing of a large amounts of information. Simultaneously, enterprises that provide these applications and services generate and analyze massive amounts of both structured and unstructured data in order to increase the quality of service to the end-user and improve the enterprises economic competitiveness. However, new challenges emerge with the high processing and storage demands. In fact, according to *IDC*, 34.7 billion gigabytes of storage were sold in the second quarter of 2016. The challenges have motivated the scientific community to focus on several research fields such as machine learning, and analytics.

Currently, companies that want to leverage big data storage and analytics, can follow two different options: (i) acquire and manage a private infrastructure, being also responsible for the internal information management; (ii) resort to cloud computing services. The first option may not be sustainable, in many cases, due to the high costs that a private infrastructure imposes, from the equipment to the manpower necessary to maintain it. In order to avoid such problems, companies can instead resort to cloud computing services, which provide a elastic and pay-as-you-go model for storage and computing power. However, this computational shift causes data control to be migrated to a third party (the cloud providers), leading to several security and privacy vulnerabilities (*e.g.*, The *iCloud* attack that revealed the private content of its clients).

Thus, in order to solve these constraints, this dissertation main goals are to study and develop new mechanisms that allow secure analytical processing of information. In detail, the following contributions are presented: a state-of-the-art study of secure analytical systems, as well as the cryptographic techniques supported by them. A new modular and flexible platform, *SafeAnalytics*, that integrates *SafeNoSQL*, a system that allows secure storage and processing of information in untrusted infrastructures, and *Apache Spark*, an analytical processing system. And, finally, a prototype evaluation using realistic workloads that shows that it is possible to leverage *SafeAnalytics* security guarantees while having a performance impact inferior to 20% compared to current solutions that not provide data confidentiality guarantees.

RESUMO

Hoje em dia é cada vez mais comum recorrermos a múltiplas aplicações e serviços online para gerir o nosso quotidiano, levando à produção de grandes quantidades de informação. Simultaneamente, as empresas que fornecem estes serviços geram e analisam quantidades massivas de informação e metadados com o objetivo de melhorar os interesses dos seus utilizadores e a sua competitividade económica. Contudo, torna-se cada vez mais difícil armazenar e processar eficientemente esta enorme quantidade informação. De facto, segundo a IDC, no segundo trimestre de 2016 foram vendidos 34.7 mil milhões de gigabytes de armazenamento. Este desafio tem desencadeado diversas contribuições em campos como *machine learning* e processamento analítico de dados.

Atualmente, existem duas opções para as empresas que querem tirar partido do armazenamento e processamento de dados: adquirir e administrar uma infraestrutura privada, assumindo a gestão interna da informação, ou recorrer a serviços de computação na nuvem. A primeira opção pode não ser a ideal devido aos elevados custos de aquisição e administração de uma infraestrutura e serviços privados. De forma a evitar este tipo de problemas, a opção de recorrer a serviços de computação na nuvem torna-se bastante atrativa devido à sua flexibilidade de armazenamento e poder computacional. Contudo, com o uso deste tipo de serviços, o controlo dos dados passa para terceiros podendo levar a falhas de segurança e de privacidade, tal como foi o caso do ataque à *iCloud* em que foi revelado conteúdo privado dos seus clientes.

Assim, de forma a resolver estas limitações, esta dissertação tem como principal objetivo estudar e desenvolver novos mecanismos que permitam o processamento analítico seguro de informação. Em detalhe, são apresentadas as seguintes contribuições: um estudo do estado da arte dos sistemas de processamento analítico seguro, bem como as técnicas criptográficas suportadas por estes. Uma nova plataforma modular e flexível de processamento analítico seguro denominada *SafeAnalytics*. Um protótipo desta plataforma que integra os sistemas *SafeNoSQL*, um sistema que permite armazenamento e processamento seguro de informação em infraestruturas não confiáveis, e *Apache Spark*, um sistema de processamento analítico. E, por fim, uma avaliação do protótipo recorrendo a cargas de trabalho realistas que mostra que é possível alavancar as garantias de segurança do *SafeAnalytics* com um impacto no desempenho inferior a 20%, quando comparado com soluções atuais que não contemplam garantias de confidencialidade de dados.

CONTEÚDO

LISTA DE FIGURAS

LISTA DE TABELAS

ACRÓNIMOS

ACLs Access Control Lists.
AES Advanced Encryption Standard.
API Application Programming Interface.
ASHE Additively Symmetric Homomorphic Encryption.
AST Abstract Syntax Tree.

CBC Cipher Block Chaining.
CTR Counter Mode.

DAG Direct Acyclic Graph.
DES Data Encryption Standard.
DSL Domain Specific Language.

FHE Fully-Homomorphic Encryption.
FPE Fully-Homomorphic Encryption.

GCM Galois Counter Mode.
GFS Google File System.

HDD Hard Disk Drive.
HDFS Hadoop Distributed FileSystem.

IBM International Business Machines.
IDEA International Data Encryption Algorithm.
IV Initialization Vector.

JDBC Java Database Connectivity.
JVM Java Virtual Machine.

MAC Message Authentication Code.

NoSQL Not Only SQL.

ODBC Open Database Connectivity.
OPE Order-Preserving Encryption.

PHE Partial-Homomorphic Encryption.

RAID Redundant Array of Independent Disks.
RC5 Rivest Chiper 5.
RDD Resilient Distributed Dataset.
RDG Resilient Distributed Graph.
RGPD Regulamento Geral de Protecção dos Dados.

SE Searchable Encryption.
SHC Spark-HBase Connector.
SPLASHE SPLayed ASHE.
SQL Structured Query Language.

XOR Exclusive Or.

YARN Yet Another Resource Negotiator.

1

INTRODUÇÃO

Com o decorrer do tempo e a evolução tecnológica, tornou-se mais comum recorrermos a múltiplas aplicações e serviços online para gerir o nosso quotidiano, levando à produção de grandes quantidades de informação. Estes dados são gerados por transações online, emails, multimédia, pesquisas, relatórios de saúde, interações em redes sociais, sensores, dispositivos móveis, entre outros. Segundo a *International Business Machines (IBM)*, todos os dias são gerados 2.5 *exabytes* de dados, sendo este valor tão elevado que 90% dos dados atuais foram criados nos 2 últimos anos[52].

Para muitas empresas, este crescimento exponencial dos dados torna inviável a aquisição e manutenção de uma infraestrutura privada para armazenar e analisar quantidades massivas de informação. Desta forma, cada vez mais os serviços de computação em nuvem são vistos como uma solução para este problema visto que os custos de aquisição e administração passam a ser efetuados maioritariamente por terceiros. Visto que a nuvem segue um modelo elástico, flexível e escalável, guardar os dados nesta traz vários benefícios, por exemplo, é fácil alocar mais recursos de forma a que as aplicações alojadas na nuvem rapidamente escalem para atender a um número elevado de clientes. A administração, compra e renovação de *hardware* deixa também de ser necessária para os clientes, e estes só pagam pelos recursos que utilizam.

1.1 PROBLEMAS E OBJETIVOS

Embora a nuvem forneça um serviço flexível e escalável com alta disponibilidade, nem tudo são ponto positivos, já que as empresas ao moverem os dados para a nuvem deixam de ter controlo direto sobre eles, ou seja, os fornecedores do armazenamento passam a ter controlo dos dados, isto significa que os fornecedores desse serviço podem efetuar processamento sobre estes e vender essa informação. Algumas dessas empresas são a *Amazon*, *Google*, *Microsoft*, entre outras[36]. Ainda, esta migração levanta também problemas para o consumidor final, na medida em que a informação pessoal fica suscetível a ataque, tal como aconteceu em 2009 à companhia *RockYou* que foi alvo de um ataque em que foram reveladas informações de pelo menos 32 milhões de contas[27].

Mais recentemente surgiu o escândalo da *Cambridge Analityca* em que foram revelados dados pessoais de mais de 87 milhões de utilizadores do *Facebook*[44]. Estima-se que estes dados foram usados para construir perfis dos utilizadores do *Facebook* para fins publicitários durante eleições, influenciando a opinião destes a favor do político que contratou o serviço[18].

Ultimamente, tem surgido cada vez mais legislação que se preocupa quanto à divulgação de informação sensível, sendo a Europa pioneira neste campo com a criação da *Regulamento Geral de Protecção dos Dados (RGPD)*[19]. A *RGPD* é um conjunto de regras sobre como as empresas devem processar os dados pessoais dos utilizadores que usam os seus serviços. Cabe às empresas assegurar a privacidade e a proteção de dados pessoais, sendo sujeitas a multas caso não respeitem os requisitos impostos pela *RGPD*.

Para colmatar alguns destes problemas foram propostos diferentes sistemas de bases de dados que fornecem soluções face aos problemas descritos anteriormente. Estes sistemas permitem que interrogações *Structured Query Language (SQL)* sejam executadas sobre os dados cifrados usando esquemas criptográficos adequados. Assim sendo, os administradores da bases de dados não têm acesso aos dados decifrados e mesmo que os servidores sejam comprometidos, o adversário não terá acesso aos dados dos utilizadores do serviço. Apesar destes sistemas conseguirem efetuar certos tipos de computação, tais como igualdades e ordenação de informação, existem poucas interrogações analíticas que caem nesta categoria, e mesmo algumas das interrogações que conseguem executar implicam um impacto no desempenho e espaço de armazenamento significativo.

Vários sistemas de processamento analítico seguro não são flexíveis o suficiente, quer a nível de operações que permitem, quer a nível de requisitos de espaço de armazenamento, limitando o seu uso em aplicações. Desta forma, esta dissertação tem como principal objetivo dotar sistemas de processamento analítico com a capacidade de efetuar processamento sobre dados cifrados, ou seja, oferecer a mesma interface que os sistemas atuais oferecem, mas adicionar uma camada de segurança, que garante a privacidade dos dados dos utilizadores. Assim, a adição de segurança é transparente para o utilizador. Este sistema deverá ser flexível ao ponto de escolher entre diversas técnicas criptográficas de forma a assegurar o melhor compromisso entre funcionalidade, desempenho e segurança.

1.2 Contribuições

Esta dissertação apresenta o *SafeAnalytics*, um sistema de processamento analítico seguro flexível e modular. De modo a ir de encontro com os objetivos propostos nesta dissertação foram feitas as seguintes contribuições:

- Um estudo de plataformas de processamento analítico, nomeadamente o *Apache Spark*, e das diferentes técnicas criptográficas necessárias para garantir a privacidade das operações efetuadas por estes sistemas.
- Uma nova plataforma de processamento analítico seguro, denominada *SafeAnalytics*, que combina diferentes técnicas criptográficas, de forma modular, garantindo novos compromissos relativamente ao desempenho, segurança e funcionalidade quando comparada com as soluções existentes.
- Um protótipo da plataforma que integra os sistemas *SafeNoSQL*, um sistema que permite armazenamento e processamento seguro de informação, e *Apache Spark*, uma plataforma de processamento analítico.
- Avaliação do protótipo recorrendo à plataforma de avaliação *TPC-DS* e a diferentes modelos de instalação do protótipo com diferentes garantias de segurança e desempenho.

Para além destas, foram feitas algumas contribuições secundárias. De forma a assegurar a correta conexão entre o sistema de processamento analítico e a base de dados NoSQL, foi usado um conetor (Spark-HBase), o qual foi submetido a algumas correções e otimizações. Primeiro, foram encontrados (e resolvidos) vários problemas quanto à tradução das interrogações (interrogações *SQL* para interrogações *Spark*, gerando por sua vez resultados incorretos. Segundo, foi feita uma extensão do conetor para suportar o sistema *SafeNoSQL* como *backend* de armazenamento de dados.

1.3 Estrutura da Dissertação

No capítulo 2 é feita uma revisão do estado da arte, sendo descritos os sistemas de processamento analítico, esquemas criptográficos e sistemas que permitem processamento analítico seguro. No capítulo 3 é apresenta a arquitetura do sistema proposto e detalha sobre o papel de cada componente. No capítulo 4 é apresentada a implementação do sistema. São também apresentadas as contribuições feitas para projetos usados na dissertação, nomeadamente o conetor *Spark-HBase* e a plataforma de avaliação usado para testar o sistema. No capítulo 5 é feita a avaliação do *SafeAnalytics* comparando-o com outros sistemas sem garantias de segurança para verificar o impacto no desempenho. Por fim, no capítulo 6 são feitas algumas observações sobre o trabalho realizado e alguns pontos que seriam interessantes de explorar como trabalho futuro.

2

ESTADO DA ARTE

O termo *Big Data* já existe desde 1990, mas décadas mais cedo as organizações já faziam análises sobre os dados, tipicamente analisando manualmente dados de tabelas para descobrir tendências e fazer previsões. Com o crescimento do universo digital, a produção de dados aumentou significativamente, tornando o armazenamento e a análise de dados uma tarefa difícil, devido à elevada complexidade e devido à elevada complexidade e falta de estruturação da informação. Para combater este problema surgiram plataformas de processamento analítico, visto que as soluções tradicionais não conseguiam dar resposta a esta evolução e era necessário identificar mudanças nas tendências para uma organização obter vantagem competitivas. Desenvolver uma infraestrutura com a habilidade de escalar, com bom desempenho, e que seja resiliente a problemas que surgem aquando da análise deste volume de dados não é uma tarefa trivial. Desta forma, houve a necessidade de desenvolver uma plataforma que conseguisse processar os dados de forma distribuída. Com este propósito, surgiu o ecossistema *Apache Hadoop*, que tem como foco principal o processamento massivo de informação num ambiente distribuído.

2.1 PLATAFORMAS DE PROCESSAMENTO ANALÍTICO

As plataformas de processamento analítico dividem-se em duas categorias: processamento em *batch* e processamento em *stream*. O processamento em *batch* tem três fases: na primeira, os dados são recolhidos, por exemplo, de aplicações durante um período de tempo; na segunda fase são processados, sendo que este processo pode demorar horas ou até dias, dependendo dos dados recolhidos na fase anterior; por fim, na terceira fase, os resultados são apresentados ao analista. Este tipo de processamento é usado quando é necessário lidar com grandes quantidades de dados ou quando as fontes de dados não são capazes de fornecer os dados em *stream*. Processamento em *batch* é ideal para situações em que não são necessários resultados em tempo real ou quando a prioridade é o volume de dados processados e não o tempo de processamento. Por outro lado, existe o processamento em *stream* que requer um contínuo fornecimento de dados e processamento constante, fornecendo análises em tempo real. Este tipo de sistema é particularmente útil em sistemas que operam em tempo real,

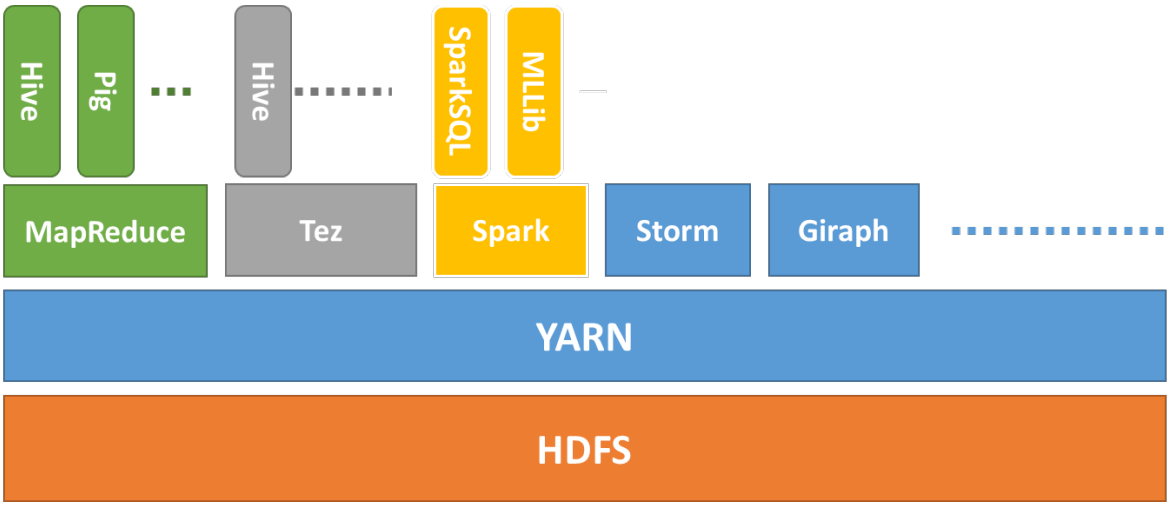


Figura 1: Arquitetura do ecossistema *Hadoop*[15]

como por exemplo, sistemas de deteção de fraude visto que conseguem detetar anomalias em tempo real e abortar as operações[12].

Nesta secção são descritas algumas das plataformas de processamento analítico distribuído mais usadas atualmente, *Hadoop MapReduce*[30] e *Apache Spark*[7].

2.1.1 *Apache Hadoop*

O *Apache Hadoop* é uma plataforma de processamento que facilita o uso de uma rede de nós de computação e armazenamento para resolver problemas que envolvam uma grande quantidade de informação e computação. Como processamento Segue o modelo *MapReduce* desenvolvido pela *Google* para processar informação de forma paralela e distribuída. O *Hadoop* é composto por três partes: a camada de armazenamento (*HDFS*), a camada de gestão de recursos (*YARN*) e a camada de processamento (*MapReduce*). Para conseguir processar informação de forma paralela e distribuída o *Hadoop* parte os ficheiros em blocos e estes são distribuídos pela rede de nós. De seguida, os executáveis são distribuídos pelos nós, que processam os dados em paralelo, tirando proveito a coalocação de dados, ou seja, não é necessário transferir os dados pela rede para serem processados num único nó. A figura 1 apresenta os várias componentes do ecossistema *Hadoop*.

Estas componentes são discutidas nas secções abaixo. No âmbito desta dissertação, daremos maior destaque ao *MapReduce*, *HDFS*, *YARN* e ao *Apache Spark*.

MapReduce

MapReduce é um modelo de computação usado para o processamento de grandes volumes de dados. Esta computação é geralmente expressa por duas funções: *map* e *reduce*, em

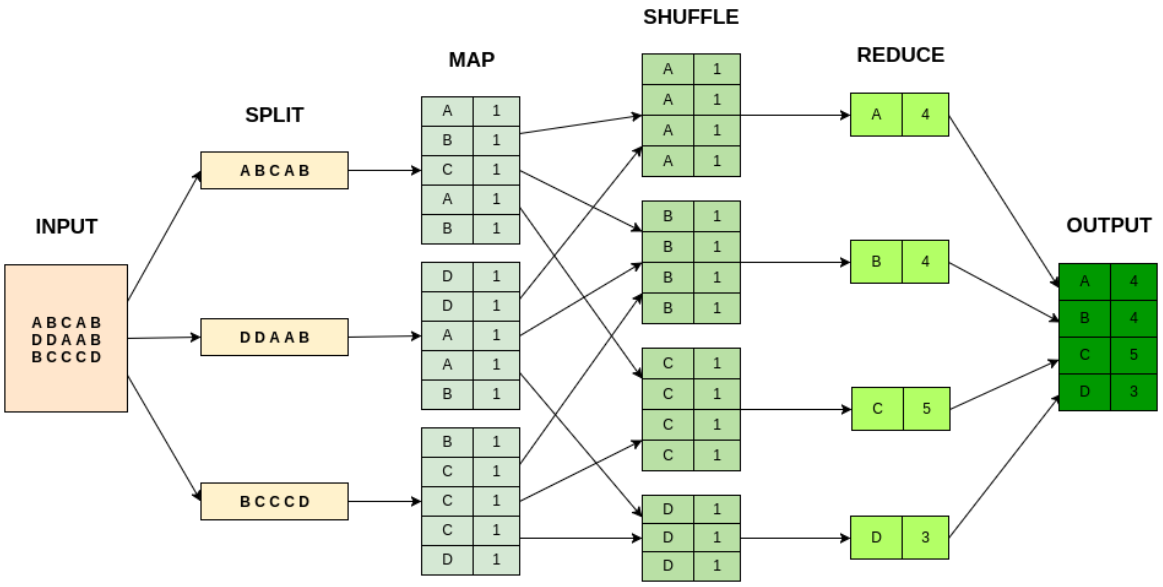


Figura 2: Fluxo da uma contagem de palavras num sistema *MapReduce*

que a função *map* recebe um par chave-valor e produz um conjunto de pares chave-valor intermédios.

A função *reduce* aceita uma chave e um conjunto de valores correspondentes a essa chave combina esses valores formando conjuntos mais pequenos. Para além do *map* e *reduce*, são também necessárias as funções *split* e o *shuffle*. *Shuffle* é o processo de agrupar os dados por chave e transferir os dados mapeados para o processo de redução, sendo que este processo de *shuffling* pode ser iniciado sem que o mapeamento tenha terminado. O *split* tem como função aumentar a paralelização do processamento dos dados, sendo isto atingido através da distribuição dos dados por diferentes nós de processamento, o que vai reduzir o tempo de processamento. Na figura 2 é apresentado o fluxo de um exemplo em que é calculado o número de vezes que cada palavra aparece num texto. Na primeira etapa os dados são separados em vários blocos que são enviados para os nós de processamento. De seguida, cada nó vai fazer o processamento que o utilizador definiu em paralelo com os outros nós. O resultado é então *shuffled* entre os nós e enviado para os *reducers*, que agregam os resultados, que são armazenados, por exemplo, no *HDFS*.

HDFS

O *HDFS*[54] é um sistema de ficheiros distribuído, inspirado no *Google File System (GFS)*[33], foi desenvolvido para armazenar e processar grandes quantidades de dados. Ao contrário de outros sistemas de ficheiros, o *HDFS* não usa mecanismos de proteção de dados, como *Redundant Array of Independent Disks (RAID)* para tornar os dados duráveis. Ao invés, replica os dados entre as nós, tendo como vantagem para além da durabilidade dos dados, a

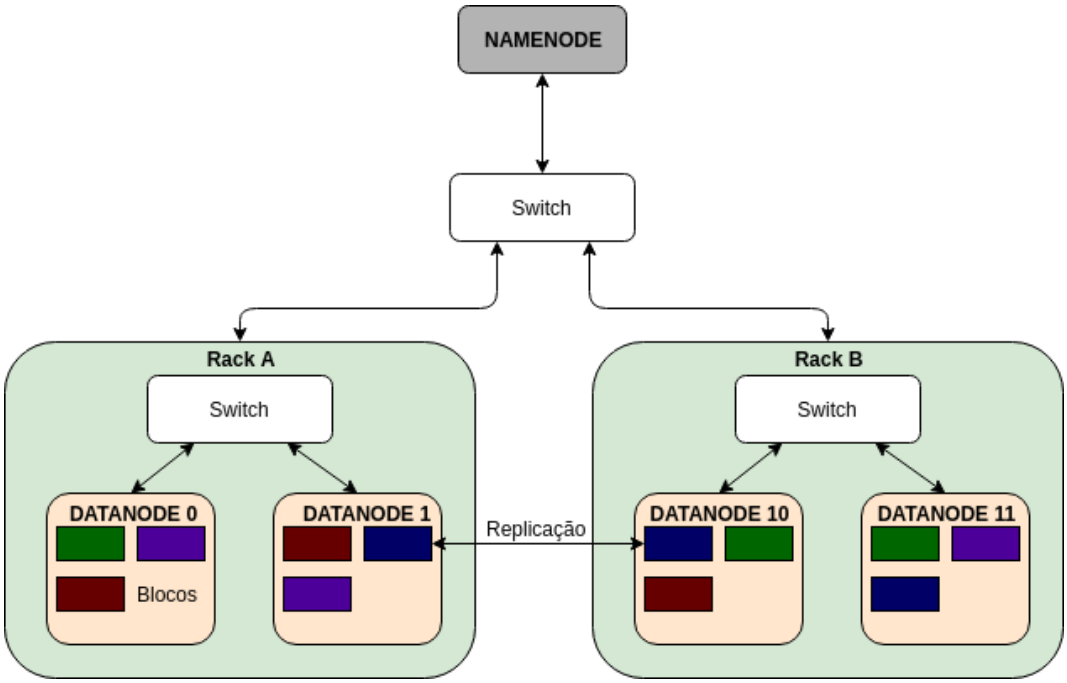


Figura 3: Arquitetura do sistema *HDFS*

possibilidade de alavancar o processamento em paralelo para aplicações a utilizar o mesmo e processar as várias cópias em paralelo. Ainda, versões mais recentes do *HDFS*, suportam também redundância por *erasure-coding*[2].

O *HDFS* segue o modelo *master/slave*, sendo o *master* do sistema denominado de *NameNode* e os *slaves* denominados de *DataNodes*, como apresentado na arquitetura do sistema na figura 3. Como é comum no modelo *master/slave*, o *NameNode* está encarregue de criar tarefas e dividi-las em várias para os *DataNodes* executarem, sendo que neste contexto, as principais tarefas são a criação, escrita, leitura e a replicação de ficheiros.

O conteúdo dos ficheiros é normalmente dividido em blocos de 128 megabytes e replicado por 3 *DataNodes*, sendo estes valores configuráveis pelo utilizador. Por cada réplica de um bloco são criados 2 ficheiros no *DataNode*, um que contém os dados e outro que contém metadados referentes a *checksums*.

De forma a otimizar o desempenho de escritas e leituras, a largura de banda utilizada e a garantir a confiabilidade dos dados armazenados, o *HDFS* utiliza diferentes políticas para a escolha dos nós em que armazena as réplicas. A política padrão oferece uma troca entre minimizar o custo de escritas, maximizar a disponibilidade e a confiabilidade dos dados, e minimizar o uso de largura de banda em leituras. Esta política segue o seguinte fluxo: quando um bloco é criado, este é armazenado no *DataNode* em que foi criado, sendo que a segunda e terceira réplicas são armazenadas numa *rack* diferente. Caso existam mais réplicas, o que depende do nível de replicação escolhido, são distribuídas pelo resto do *cluster*, não sendo possível armazenar mais do que 2 réplicas na mesma *rack*, nem armazenar mais do

que 1 réplica no mesmo *DataNode*. Uma prática comum é dividir os nós por múltiplas *racks*, visto que não é prático conectar todos os nós ao mesmo ponto, ou seja, para criar uma topologia, o *cluster* é dividido em *clusters* mais pequenos, sendo estes denominados de *racks* que são conectadas por *switches*.

Para monitorizar os *DataNodes* que estão disponíveis e obter informações sobre estes, o *NameNode* usa *heartbeats*, que são sinais usados para estes comunicarem. O intervalo normalmente usado entre *heartbeats* é de 3 segundos, sendo que se um *NameNode* deixar de receber *heartbeats* de um *DataNode* durante 10 minutos, é considerado fora de serviço e é iniciado um novo processo de replicação dos blocos que estavam no *DataNode* que deixou de operar. Outra utilidade dos *heartbeats* é a troca de mensagens entre o *NameNode* e os *DataNodes*, em que o *NameNode* pode submeter comandos como: apagar as réplicas de blocos, ligar/desligar os *DataNodes*, replicar blocos para outros *DataNode*, entre outros.

De modo a interagir com o sistema de ficheiros, os clientes recorrem às interfaces que o *HDFS* fornece, nomeadamente a interface *Java native, web*, terminal, entre outras. Tal como sistemas convencionais, estas interfaces permitem submeter operações comuns dos sistemas de ficheiros, como criar, ler, escrever e apagar ficheiros e diretorias. Quando uma aplicação necessita de efetuar uma leitura esta processa-se da seguinte forma: a aplicação pede ao *NameNode* uma lista de *DataNodes* que contêm blocos do ficheiro que é necessário ler; de seguida a aplicação contacta o *DataNode* diretamente para que o bloco seja transferido. Quando se trata de uma escrita, a aplicação contacta o *NameNode* para que este lhe dê uma lista de *DataNodes* disponíveis para escrita. Quando a escrita de um bloco do ficheiro terminar, a aplicação contacta de novo o *NameNode* para lhe fornecer uma nova lista de *DataNodes* para o próximo bloco. Os *DataNodes* estão encarregues de confirmar à aplicação o sucesso da escrita dos blocos, que por sua vez notifica o *NameNode*, sendo que no caso da existência de alguma falha na confirmação da escrita, o bloco é replicado para outro *DataNode*.

Para garantir a escalabilidade, desempenho, disponibilidade e integridade dos dados o *HDFS* impõe algumas limitações em comparação com os sistemas de ficheiros locais:

- **IMUTABILIDADE DOS FICHEIROS:** Não é possível alterar nenhuma parte de um ficheiro que já esteja escrito, visto que o *HDFS* é um sistema de ficheiros *append-only*. A alternativa é reescrever o ficheiro com as novas alterações e apagar o antigo, o que é ineficiente. Este modo de funcionamento é conhecido como *write-once-read-many*.

- **COERÊNCIA DOS DADOS:** Durante as operações de escrita o ficheiro não fica visível para os outros clientes. Apenas quando o cliente terminar a operação de escrita é garantido que os outros clientes tenham acesso ao ficheiro completo.

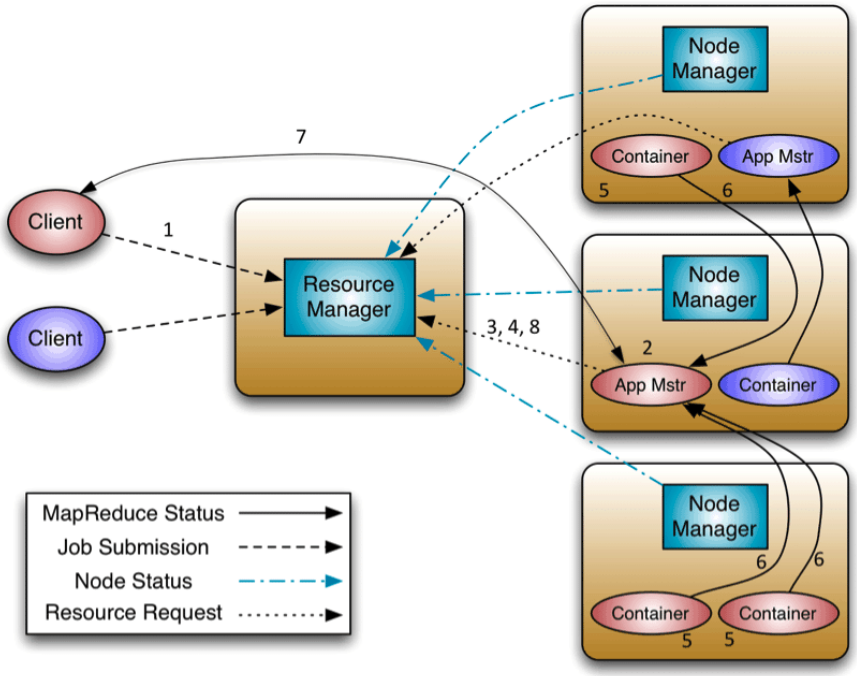


Figura 4: Arquitetura do sistema *YARN*[1]

- **NÍVEIS DE ACESSO:** Nos sistemas de ficheiros *Unix* existem três tipos de permissões: escrita, leitura e execução. No *HDFS* a permissão de execução não é utilizada visto que não é possível a execução de ficheiros.

Em suma, o *HDFS* é um sistema de ficheiros que deve ser utilizado quando é necessário guardar uma grande quantidade de informação e é necessário aceder em paralelo aos dados nele guardado, não sendo a opção ideal se for necessário armazenar ficheiros pequenos ou modificar dados existentes[43].

YARN

O *YARN* é um gestor de recursos de *clusters*. Originalmente, a gestão de recursos e o escalonamento era feito pela mesma componente que tratava do processamento de dados, mas recentemente, com o desenvolvimento do *YARN* estas capacidades foram desacopladas da componente de processamento, permitindo que o *Hadoop* desse suporte a outras técnicas de processamento e aplicações.

O *YARN* é composto por três componentes: o *ResourceManager*, o *NodeManager* e o *Application Master*, tal como apresentado na figura 4.

1 Retirado de *"Apache Hadoop Yarn"*[4]

O *Resource Manager* é um *daemon* global ao sistema, principalmente responsável por escalonar os recursos no sistema entre as aplicações concorrentes, otimizando a utilização de recursos do sistema. Este processo gere quantos nós e recursos estão disponíveis e coordena a distribuição de recursos pelas aplicações submetidas pelos utilizadores. Por ter uma visão global do *cluster* o *Resource Manager* consegue tomar decisões de alocação de recursos de acordo com prioridades de aplicações, filas de espera, *Access Control Lists (ACLs)*, localidade dos dados, entre outros.

Quando um utilizador submete uma aplicação, um processo denominado *Application Master* é iniciado para coordenar a execução das tarefas da aplicação. Isto inclui, monitorizar tarefas, reiniciar tarefas com falhas e calcular os recursos utilizados pela aplicação. Cada aplicação submetida tem um *Application Master* responsável por negociar recursos com o *Resource Manager*, sendo que os recursos alocados para um *Application Master* são denominados de *containers*.

Por fim, o *Node Manager* é responsável por lançar e gerir os *containers* num nó, monitorizando os recursos utilizados e enviando essa informação ao *Resource Manager*. Os *containers* executam tarefas especificadas pelo *Application Master* criando um *container* para cada tarefa. A execução de uma aplicação, utilizando a figura 4 como referência, segue o seguinte fluxo:

1. O utilizador submete uma aplicação.
2. O *Resource Manager* determina o nó em que deve criar um *container* para o *Application Master* e inicia-o.
3. O *Application Master* é registado no *Resource Manager*, permitindo à aplicação comunicar diretamente com o *Application Master*.
4. O *Application Master* negoceia a alocação de recursos apropriados para os *containers* com o *Resource Manager*.
5. Após a alocação dos recursos para os *containers*, o *Application Master* lança-os seguindo a especificação do *Node Manager*.
6. A aplicação é executada no *container* enviando informação sobre o progresso ao *Application Master*.
7. Durante a execução da aplicação, o cliente comunica diretamente com o *Application Master* para obter informação sobre o progresso da aplicação.
8. Assim que a aplicação termina, o *Application Master* é removido da lista de aplicações em execução do *Resource Manager* e o processo termina, libertando os recursos utilizados.

Em suma, o *YARN* elimina a necessidade de existir um cliente ativo, ou seja, o processo que inicia a aplicação pode terminar visto que a coordenação é feita pelo *YARN*. Para além disto, permite a alocação de recursos entre diferentes plataformas que utilizam o *YARN* sem que seja necessário alterar configurações.

2.1.2 *Apache Spark*

De forma a colmatar alguns problemas do *Hadoop MapReduce*, tal como processamento em tempo real, surgiu o *Apache Spark*. Esta plataforma de computação distribuída baseada no *Hadoop MapReduce* estende o modelo computação de forma a suportar mais tipos de operações, o que inclui interrogações interativas e processamento em *stream*.

O *Apache Spark* é construído sobre os conceitos de *Resilient Distributed Dataset (RDD)*, que representam uma coleção de objetos imutáveis distribuídos, divididos em partições lógicas distribuídas pelos diferentes nós de um *cluster*, e *Direct Acyclic Graph (DAG)s* compostos por vértices e arestas, sendo que representam *RDDs* e as operações aplicadas sobre estes respetivamente. Estas estruturas têm duas propriedades que permitem otimizações ao processamento. Primeiro, são grafos acíclicos, ou seja, uma vez que um *RDD* seja transformado não é possível voltar ao estado anterior. Segundo, são grafos diretos, ou seja, todos os nós são ligados, criando uma sequência de operações sobre as estruturas de dados. Estas estruturas, para além de permitirem otimizações às interrogações dos clientes, oferecem também tolerância a falhas, visto que se trata de um sistema de processamento distribuído sujeito a falhas nos nós de computação. Caso ocorra uma falha num nó em que esteja a decorrer computação, o gestor de recursos do sistema vai verificar que o nó não concluiu a tarefa que lhe foi atribuída com sucesso e esta vai ser reatribuída a outro nó para que o processamento prossiga, sem que tenha que ser reiniciado.

Um *RDD* é dividido em várias partições distribuídas pelos nós de um *cluster*. Geralmente, um maior número de partições mais pequenas permite que o processamento seja distribuído entre mais *workers*, mas um menor número de partições maiores permite que o processamento seja concluído mais rapidamente visto que não existe tanta penalização na serialização e transmissão de dados através da rede. Por regra geral, o número de partições deve ser pelo menos igual ao número de *executors* do *cluster*, de forma a aproveitar ao máximo o paralelismo.

Tal como alguns dos sistemas falados anteriormente, o *Spark* segue uma arquitetura *master/slave*. A figura 5 apresenta a arquitetura do *Spark* com um gestor de recursos. O *Driver* é responsável pela calendarização da execução de tarefas, bem como negociar recursos com o *Resource Manager*, uma componente do *YARN*. Esta componente também é responsável pelo armazenamento dos metadados dos *RDDs* e das suas partições, bem como a criação dos grafos de execução(*DAGs*) e do particionamento da atividade em múltiplas etapas.

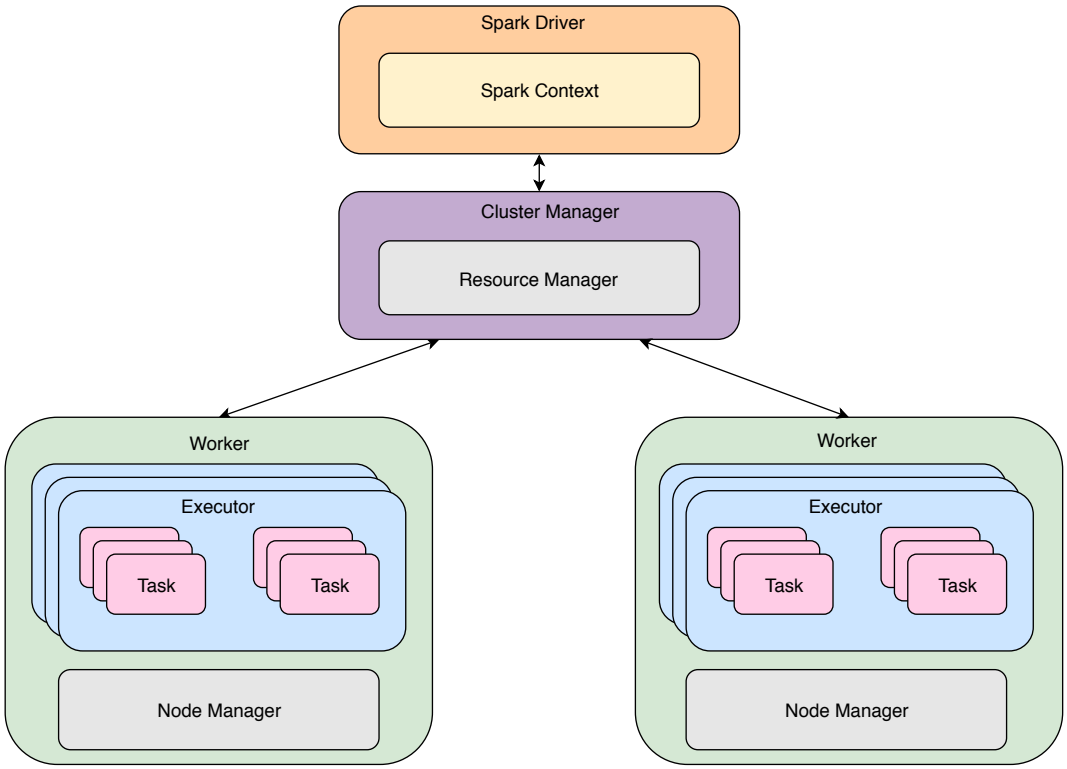


Figura 5: Arquitetura do sistema *Apache Spark* (com um gestor de *cluster*)

O *Cluster Manager* lida com a criação de processos para execução das atividades, bem como a gestão de recursos do *cluster*. O *Spark* suporta três gestores de *cluster*, cada um com as suas respetivas vantagens e desvantagens, nomeadamente o *Standalone Cluster Manager*, *Hadoop YARN* e o *Apache Mesos*. O *Spark* possui um gestor de *cluster* incorporado, o *Standalone Cluster Manager*, que é ideal para executar aplicações simples num ambiente distribuído. Por padrão este gestor aloca todos os recursos do sistema o que faz que não seja ideal para um ambiente em que é necessário correr diferentes aplicações, assim este gestor é predominantemente utilizado em ambientes de teste. Para casos em que é necessário alocar recursos de forma dinâmica existem duas opções: o *YARN* e o *Mesos*. A principal diferença entre estes dois gestores de recursos do *cluster* está na forma como atribuem recursos para os processos. Com o *Mesos*, quando uma atividade é submetida, é enviado um pedido ao *Mesos master* que determina os recursos disponíveis no *cluster*, e faz uma oferta que pode ser aceite ou rejeitada pelo *Driver*. Isto permite ao sistema alocar da melhor forma uma atividade aos recursos disponibilizados, enquanto que no *YARN* a atribuição de atividades a recursos é feita por este, sem que possa ser rejeitada pelo *Driver*.

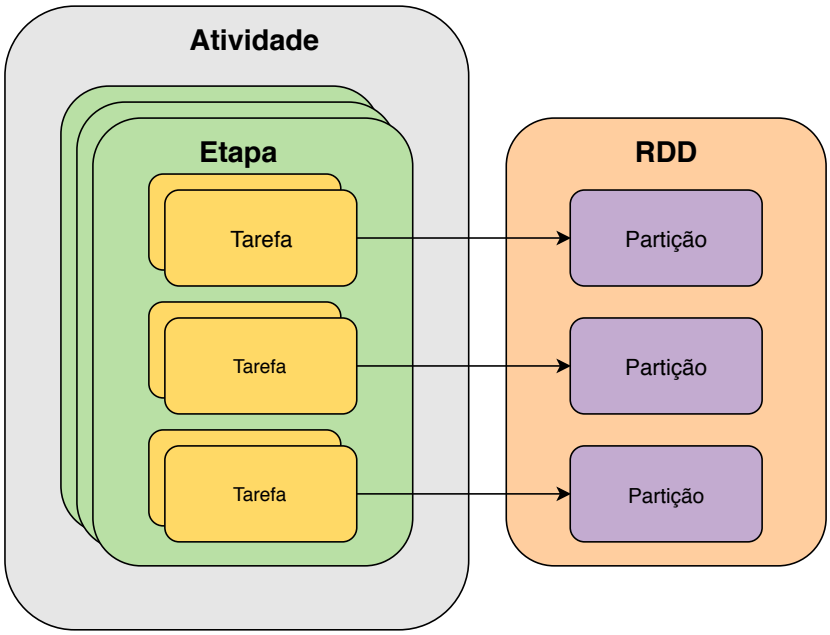


Figura 6: Atividades, Etapas e Tarefas

Plano lógico e plano físico de execução

Uma das técnicas usadas pelo *Apache Spark* para ter vantagem em relação a outras plataformas de processamento analítico baseia-se em *DAGs*. Esta estrutura é composta por vértices e arestas, sendo que representam *RDDs* e as transformações aplicadas sobre estes respetivamente. Assim é criada uma linhagem de *RDDs* que dá origem ao plano lógico de execução. Visto o *Spark* ser implementado com *Scala*, este retém algumas características da linguagem, nomeadamente a *laziness*. Assim, este plano lógico só é executado quando é realizada uma ação que necessite de dados, sendo otimizado e convertido num plano físico de execução.

Um plano físico de execução é constituído por etapas. É representado por um *DAG*, responsável por guardar informação acerca das operações aos *RDDs*, que oferece a tolerância a falhas referida anteriormente, ou seja, caso falhe uma etapa esta é reatribuída a outro nó continuar a computação. Por padrão no fim de cada etapa os dados são transmitidos para outros nós que necessitam destes para continuar o processamento.

Atividades, Etapas e Tarefas

Como referido anteriormente, o *Spark* apenas executa as transformações quando existir uma ação que obriga os dados a serem transferidos para o *Driver*. Assim, é criada uma atividade associada ao plano físico de execução, representando todas a transformações e ações a serem feitas sobre os dados.

Na figura 6 é apresentado um esquema com as relações entre atividades, etapas e tarefas com *RDDs*. Conforme detalhado, uma atividade é dividida em diferentes etapas que são constituídas por um conjunto de tarefas. Cada tarefa é aplicada sobre uma partição diferente, ou seja, são executadas em paralelo computando resultados parciais de cada partição calculando o agregado no final da etapa se necessário.

Spark SQL

De forma a agilizar a construção de aplicações sobre o *Spark* foram desenvolvidas várias ferramentas que criam um nível de abstração sobre a *Application Programming Interface (API)* base do *Spark*.

A computação na plataforma *Hadoop MapReduce* é dividida em múltiplas iterações e cada uma destas normalmente realiza leituras do *HDFS*, aplica as funções *map* e *reduce* e escreve o resultado de novo no *HDFS*. Visto que cada iteração é independente das outras, torna-se impossível ter conhecimento global para a otimização do processamento, o que faz com que algoritmos iterativos sejam ineficientes, visto que estão a escrever e a ler dados continuamente no *HDFS*. Plataformas de processamento como o *Spark* melhoram a desempenho visto que consideram um plano de execução global, podendo otimizá-lo e remover a necessidade de escrita e leitura contínua após cada passo de execução.

De forma a integrar a programação declarativa oferecida pelo *SQL* com a *API* de programação funcional do *Spark* surgiu o *Spark SQL*[22]. Em comparação com os sistemas anteriores, como o *Shark*[59], o *Spark SQL* oferece uma *API* declarativa que permite processamento relacional, oferecendo benefícios como a otimização automática de interrogações e permitir aos utilizadores combinar interrogações relacionais complexas e operações de processamento analítico. Para além disto o *Spark SQL* suporta diversas formas de interação com os dados, desde à utilização de *APIs* disponíveis em *Scala*, *Java* e *Python*, à utilização de conetores amplamente utilizados, como *Java Database Connectivity (JDBC)* e *Open Database Connectivity (ODBC)*.

Spark Streaming

Para efetuar processamento em *stream* foi criado o *Spark Streaming*[60]. Esta componente consome dados provenientes de fontes como o *Kafka*[6], *Flume*[3], *Kinesis*[1], que são sistemas responsáveis por recolher, agregar e mover grandes volumes de dados, conseguindo processar os dados recorrendo a funções de alto nível, como *map*, *reduce*, *join* e *window*. Os resultados podem ser armazenados em sistemas de ficheiros, por exemplo no *HDFS*, em bases de dados ou apresentados aos utilizadores. Internamente o *Spark Streaming* recebe um *stream* contínuo de dados e divide-o em partes, que são então processadas pelo motor referido anteriormente para gerar uma *stream* de resultados. O *Spark Streaming* oferece uma abstração para representar uma *stream* contínua de dados denominada *DStream* que podem

ser criadas pelas fontes de dados referidas anteriormente ou pela aplicação de funções de alto nível a outras *DStreams*, sendo que internamente são representadas por uma sequência de *RDDs*.

MLlib

O *Apache MLlib*[40] é uma plataforma distribuída de *machine learning* construída sobre o *Spark Core*, aproveitando a sua arquitetura baseada em memória distribuída (*RDDs*) para aumentar o seu desempenho. Esta plataforma possui uma biblioteca que inclui algoritmos de classificação, regressão, *clustering*, e outras técnicas de otimização. Assim, visto que o *Spark* foi desenhado com o objetivo de tornar computação iterativa mais eficiente, é possível implementar algoritmos de *machine learning* de larga escala, visto que este são tipicamente de natureza iterativa.

GraphX

O *GraphX*[58] é a componente do *Apache Spark* responsável pela computação sobre grafos. Apresenta uma abstração sobre *RDDs* denominada *Resilient Distributed Graph (RDG)s* que simplifica a construção e a computação sobre grafos. Para além disso implementam *APIs*, nomeadamente para *PowerGraph* e *Pregel*, baseadas em *RDGs* de forma a facilitar a sua integração em sistemas existentes.

2.1.3 *Discussão*

Tal como discutido nesta secção, existem vários sistemas de processamento analítico que apresentam as suas vantagens e desvantagens. Para esta dissertação como plataforma de processamento analítico, vai ser usado o *Apache Spark* visto que é bastante adotado por ambas comunidades académicas e industriais, e também por pertencer ao ecossistema *Hadoop*, facilitando a sua integração com outros serviços pertencentes ao ecossistema. Tipicamente, o *Apache Spark* recorre ao *HDFS* como camada de armazenamento. Contudo, através de conetores, é possível recorrer a outras soluções de armazenamento, por exemplo, *HBase*[5]. Como um dos objetivos da dissertação é integrar o *Apache HBase* no sistema desenvolvido, vamos recorrer a um conetor desenvolvido pela *Hortonworks*[13], o *Spark HBase Connector*[37] de forma a facilitar esta integração.

2.2 Esquemas Criptográficos

Com o decorrer dos anos, surgiram novos esquemas criptográficos para responder aos problemas de segurança, privacidade e confidencialidade nas diversas áreas de tecnologias

de informação. Recentemente, com o aumento da produção de dados e com a necessidade de proteger a privacidade dos utilizadores, surgiu o paradigma de processamento seguro que recorre a técnicas criptográficas para proteger os dados e permite processamento sobre estes. Neste secção é feito um levantamento dos esquemas mais utilizados nos sistemas de processamento seguro atuais.

2.2.1 *Cifras simétricas e assimétricas*

Os algoritmos que usam a mesma chave para cifrar uma mensagem e decifrar o criptograma denominam-se por algoritmos de chave simétrica. Estes algoritmos são normalmente utilizados para garantir autenticação, integridade e confiabilidade dos dados, sendo que no âmbito desta dissertação são utilizados para proteger os dados num servidor não confiável, em que apenas o utilizador tem acesso à chave para os cifrar e decifrar. Os algoritmos de chave assimétrica usam um par de chaves, uma chave pública, que é partilhada, e uma chave privada que apenas é conhecida pelo seu proprietário. Neste esquema qualquer interveniente pode cifrar mensagens usando a chave pública do destinatário, e apenas esse a pode decifrar, utilizando a sua chave privada. Devido há complexidade na computação deste algoritmo, este não é frequentemente utilizado diretamente para proteger dados, mas para trocas de chaves, assinaturas digitais, entre outros. Assim sendo, o foco está sobre as mais utilizadas, as cifras simétricas. Estas podem ser implementadas como cifras por bloco ou cifras sequenciais.

Cifras Sequenciais

As cifras sequenciais cifram a mensagem carácter a carácter recorrendo a um gerador de chave (interno) pseudo-aleatório. Para a implementação ser considerada segura, o resultado do gerador pseudo-aleatório deve ser imprevisível e o tamanho da chave usada deve ser o maior possível, visto que esta servirá de semente para o gerador. Ainda, a chave também nunca deve ser reutilizada, devido à possibilidade de ataques de reconhecimento de padrões[34].

Cifras por Bloco

Para além das cifras sequenciais, existem também as cifras por bloco. Estas processam a mensagem em blocos de bits de tamanho fixo (normalmente 128 ou 256 bits), dependendo do modo de funcionamento. As mensagens passam por uma camada de partição, produzindo um conjunto de blocos. Caso seja necessário, é aplicado *padding* ao último bloco. O *padding* é um mecanismo que adiciona *bits* arbitrários a um bloco para garantir que todos os blocos assumem sempre o mesmo tamanho. Assim, este mecanismo estende o último bloco da mensagem com *zero-bits*, sendo este esquema conhecido como *padding method 2*[20]. Para

decifrar a mensagem, aplica-se o processo inverso e revertem-se os processos de partição e *padding*, de forma a obtermos a mensagem original.

Para além de garantirem autenticidade e confidencialidade, estas cifras, permitem a construção de sistemas como o *Message Authentication Code (MAC)*, que para além de garantir autenticidade, garante a integridade de uma mensagem. O *MAC* funciona de forma semelhante a uma função de *hash*, dependendo o seu resultado de uma chave e da mensagem, permitindo detetar alterações na mensagem.

Estas cifras utilizam vários modos de funcionamento, em que são oferecidos diferentes critérios de segurança, eficiência, débito, propagação e tolerância a erros. Existem vários modos de funcionamento, o *Cipher Block Chaining (CBC)*, o *Counter Mode (CTR)* e o *Galois Counter Mode (GCM)*, entre outros, sendo os mais utilizados o *CBC* e o *GCM*[9].

O *CBC* é o modo de funcionamento mais comum. Neste modo, cada bloco da mensagem é *Exclusive Or (XOR)ed* com o criptograma gerado pela cifragem do bloco da mensagem anterior. Deste modo, cada bloco do criptograma depende dos blocos da mensagem processados até esse ponto. Para que cada criptograma seja único, é usado um *Initialization Vector (IV)*, que é *XORed* com o primeiro bloco da mensagem. Este modo prevê ainda a utilização de um *MAC* para garantir integridade das mensagens.

Quanto à eficiência, este modo não permite a paralelização da cifragem, visto que para um bloco ser cifrado necessita do bloco do criptograma anterior, o que obriga a uma execução sequencial. A operação de decifragem, por outro lado, é paralelizável, visto que cada bloco do criptograma apenas necessita do bloco do criptograma anterior. Dependendo do *IV* ser ou não aleatório, este esquema pode assumir propriedades determinísticas, ou seja, a mesma mensagem cifrada várias vezes resulta sempre no mesmo criptograma.

O modo *CTR* transforma a cifra por blocos numa cifra sequencial através de um contador. À função de cifragem é lhe aplicado um *IV*, que é conjugado com o contador. Assim, a cada cifragem, o contador é incrementado produzindo um valor diferente para cada bloco, ou seja, seria semelhante a usar um *IV* único para cada bloco. Quanto à eficiência, visto que para cifragem e a decifragem não existem dependências entre os blocos da mensagem e do criptograma é possível paralelizar estas operações.

O modo *GCM*, tal como o *CTR* é paralelizável. À função de cifragem é lhe passado um contador, que é incrementado cada vez que é usado, que é cifrado. Este contador cifrado é então *XORed* com a mensagem formando o criptograma. Este vai ser *XORed* com um código de autenticação de forma a produzir uma *tag* de autenticação, com a função de ser usada para a verificação da integridade da mensagem. A mensagem cifrada contém o *IV*, o criptograma, e o código de autenticação. Prevê-se que este modo de funcionamento vá substituir o modo *CBC* usado no *OpenSSL*[38].

Existem vários algoritmos que implementam cifras por bloco, sendo os mais conhecidos: *Advanced Encryption Standard (AES)*, *Data Encryption Standard (DES)*, *International Data En-*

cryption Algorithm (IDEA), *Rivest Chiper 5 (RC5)* e *Blowfish*[28][29][24][51][53]. O algoritmo mais usado é o *AES* operando com blocos de 128 a 256 bits e chaves de 128, 192 ou 256 bits. De forma a uniformizar os termos usados em relação às técnicas criptográficas, vai ser chamada *Standard Encryption* a uma cifra aleatória (por exemplo, AES-128 *CBC* com *IV* aleatório) e *Deterministic Encryption* a uma cifra com propriedades determinísticas (por exemplo, *AES-128 CBC* com *IV* fixo).

2.2.2 *Order-Preserving Encryption*

Os esquemas criptográficos tradicionais têm a limitação de não preservarem a ordem entre dois criptogramas, obrigando a que os dados sejam transferidos para o cliente, onde são decifrados e comparados pelo grau de grandeza. Este processo faz com que o desempenho do sistema diminua, tornando esta abordagem inviável para sistemas de processamento analítico em tempo real. De forma a resolver este problema surgiu o *Order-Preserving Encryption (OPE)*, criado por *Agrawal*[21]. O *OPE* é um esquema criptográfico determinístico que permite preservar a ordem numérica das mensagem quando cifradas. Dadas duas mensagem m_1 e m_2 e uma chave k:

$$OPE_{Enc}(k, m_1) > OPE_{Enc}(k, m_2) \; sse \; m_1 > m_2$$

em que $OPE_{Enc}(k, m)$ é a operação de cifragem. Assim, o *OPE* permite realizar operações de comparação diretamente sobre os dados cifrados, ou seja, é possível processar interrogações de igualdade, bem como as operações de ordem. Contudo este esquema não permite operações como *SUM* e *AVG*, sendo necessário passar os dados para o cliente onde é feita a computação ou usar um esquema que permita estas operações, por exemplo, *Paillier*. Contudo, para além de revelar a igualdade e a ordem dos criptogramas, em algumas implementações o esquema revela parcialmente o conteúdo da mensagem original[49].

2.2.3 *Searchable Encryption*

A pesquisa de palavras específicas numa base de dados é uma prática comum em múltiplas aplicações e serviços. Contudo, esta tarefa não é trivial (por vezes impossível) quando recorremos ao paradigma de processamento seguro. Para colmatar esta necessidade surgiu o *Searchable Encryption (SE)*[55], que permite que sejam feitas pesquisas de palavras sobre conteúdo cifrado. Originalmente, existem duas abordagens para este esquema, uma que permite a pesquisa na base de dados sequencialmente, e outra que constrói uma tabela em que são guardadas as palavras-chave e uma lista de localizações destas nos documentos.

Na primeira abordagem, para um dado documento a ser guardado no servidor, cada palavra é cifrada com uma cifra determinística, permitindo a igualdade entre criptogramas. Para ser feita uma pesquisa de uma palavra, o cliente cifra a palavra que pretende procurar com a mesma chave que foi usada para cifrar os documentos, e envia-a para o servidor. Após a receção desta por parte do servidor, é feita uma pesquisa sequencial em todos os documentos cifrados. Quando este processo terminar são enviados todos os documentos em que foi encontrada a palavra para o cliente, que decifra os documentos.

Na segunda proposta, para cada documento a ser armazenado, o cliente define as palavras-chave associadas a este. O documento é então cifrado e enviado para o servidor e cada palavra-chave definida guarda a sua localização. Da mesma forma que a proposta anterior, quando o cliente pesquisa uma palavra, esta é cifrada e enviada para o servidor. Após a receção desta, o servidor apenas tem que ir à tabela construída com as palavras-chave que o cliente definiu e devolver todos os documentos que a contém. Comparativamente à primeira abordagem, esta proposta admite um melhor desempenho, visto que não é necessário efetuar uma procura sequencial. Contudo, sempre que adicionado, removido ou atualizado um documento, há uma sobrecarga para a atualização da tabela.

A nível de segurança, a primeira abordagem, revela a igualdade entre as palavras, visto que cada uma destas é cifrada por uma cifra determinística. Na segunda abordagem são apenas reveladas as relações entre criptogramas, ou seja, se fosse feita a procura por uma palavra, o atacante ganhava a informação que vários criptogramas partilhavam essa palavra. Nesta abordagem, um atacante teria que seguir uma paradigma de criptoanálise para conseguir informação relevante sobre os criptogramas, o que não é uma tarefa trivial.

2.2.4 *Homomorphic Encryption*

Esquemas criptográficos de *Homomorphic Encryption* permitem a realização de operações arbitrárias sobre os criptogramas, permitindo ao servidor processar os dados cifrados. Estes esquemas são divididos em duas categorias: *FHE(Fully-Homomorphic Encryption)* e *PHE(Partial-Homomorphic Encryption)*. *FHE* suporta múltiplas operações algébricas , por exemplo, adições, subtrações, multiplicações, *XOR*[32]. Contudo, à medida que o conjunto de operações permitidas aumenta, o desempenho do esquema criptográfico é severamente prejudicado. Por exemplo, a implementação de *Coron*[25], necessita de transferir 73TB de informação cifrada para processar 4MB de mensagem. Posto isto, *FHE* não é um esquema apropriado para proteger informação e permitir processamento sobre criptogramas em tempo real.

Por sua vez, *Partial-Homomorphic Encryption (PHE)* relaxa o modelo de computação em relação ao *Fully-Homomorphic Encryption (FHE)*, focando-se num conjunto mais limitado de operações algébricas, por exemplo, adições, multiplicações ou *XOR*. Um exemplo deste

esquema criptográfico deste tipo é *Paillier Encryption*[45] que permite algumas operações sobre mensagens cifradas. Este esquema assume duas propriedades homomórficas, adição homomórfica de mensagens e multiplicação homomórfica de mensagens. Dadas duas mensagens m_1 e m_2 e uma chave k, este esquema assume as seguintes propriedades:

$$m_1 = HOM_{Dec}(k, HOM_{Enc}(k, m_1))$$

$$m_2 = HOM_{Dec}(k, HOM_{Enc}(k, m_2))$$

$$m_1 + m_2 = HOM_{Dec}(k, HOM_{Add}(HOM_{Enc}(k, m_1)), HOM_{Enc}(k, m_2))))$$

em que $HOM_{Enc}(k, m)$ é a operação de cifragem, $HOM_{Dec}(k, m)$ é a operação de decifragem e $HOM_{Add}(m_1, m_2)$ é a operação de adição. Visto que este esquema permite algumas operações algébricas, é muito adotado em sistemas de processamento analítico seguro[57], sendo que alguns usam variantes deste esquema otimizadas[46].

2.2.5 *Format-Preserving Encryption*

Os esquemas tradicionais garantem a privacidade dos dados, cifrando os dados sensíveis, porém, estes esquemas alteram a estrutura da base de dados. Esta limitação impossibilita a restrição do tamanho e do formato do *input*, como é o caso dos inteiros.

De forma a resolver este problema foi proposto o esquema *Fully-Homomorphic Encryption (FPE)* por *Black*[23]. A ideia por trás da criação deste esquema é a geração de criptogramas com o mesmo formato e domínio que a mensagem. Ou seja, os dados cifrados mantêm o formato e o comprimento do conteúdo original quando cifrados, sendo isto útil quando existem restrições no comprimento ou no formato dos dados. Por exemplo, assumindo que pretendemos proteger uma coluna de uma base de dados, relativa a um ano, o uso de uma cifra determinística, por exemplo *AES-128* o criptograma terá obrigatoriamente blocos de 128 *bits*, enquanto que o com o uso de *FPE* o criptograma terá 4 dígitos, preservando também o tipo da coluna. Contudo, por permitir a preservação do tipo e do tamanho da mensagem, existe um compromisso de segurança, sendo o esquema suscetível a ataques de dedução e força bruta.

2.2.6 *Discussão*

Feito o levantamento dos esquemas criptográficos de interesse para a construção de um sistema de processamento analítico seguro, é feita uma discussão em que são expostas as operações que cada esquema criptográfico permite, bem como as fraquezas de cada um.

Esquema Criptográfico	Operações	Fraquezas
Standard	Nenhuma	Nenhuma
Deterministic	Igualdades	Duplicados
Order-Preserving	Ordem e Igualdade	Duplicados, ordem e informação parcial da mensagem[2]
Partial-Homomorphic	Algébricas	Nenhuma
Format-Preserving	Igualdades	Ataques de dedução, força bruta e duplicados
Searchable	Pesquisa de palavras	Duplicados

Tabela 1: Caracterização dos esquemas criptográficos em relação às operações permitidas e as respetivas vulnerabilidades

Entende-se por fraquezas, as vulnerabilidades a ataques e fugas de informação que cada esquema apresenta.

Como foi falado ao longo deste capítulo, existem vários esquemas criptográficos que permitem que processamento analítico seja efetuado de forma segura. Na tabela 1 são descritas as operações que cada esquema permite e as suas fraquezas. As operações apresentadas são a igualdade, ordem, adição, pesquisa entre criptogramas.

A igualdade, ordem e adição referem-se à comparação da igualdade e ordem entre dois criptogramas, respetivamente, enquanto que a adição refere-se à adição de dois criptogramas. A pesquisa, tal como foi referido anteriormente, refere-se à procura de uma palavra ou de prefixos/sufixos num criptograma. A fuga de informação sobre dados duplicados acontece quando é usada a abordagem em que cada palavra é cifrada com uma cifra determinística. Por fim, o esquema *FPE* permite a verificação de igualdades entre criptogramas e poupar espaço em comparação com outros esquemas. Contudo, visto que este esquema preserva o tamanho e o tipo do criptograma, criptoanálise é um modo de ataque eficiente contra algumas implementações deste esquema[31].

2.3 Soluções de processamento analítico seguro

Já existem vários sistemas que permitem o processamento analítico seguro sobre dados sensíveis. Neste capítulo são expostos os principais sistemas, descrevendo as noções gerais de cada sistema, a sua arquitetura, o tipo de operações que realizam e os esquemas criptográficos usados.

Para processar grandes quantidades de dados torna-se extremamente desafiante sem recorrermos a ferramentas é impensável processá-los sem o uso de uma ferramenta de processamento analítico. De modo a proteger os dados e permitir ao mesmo tempo o seu processamento, foram construídos vários sistemas de processamento seguro. Para além de

2 A revelação parcial da mensagem depende da técnica criptográfica utilizada[49]

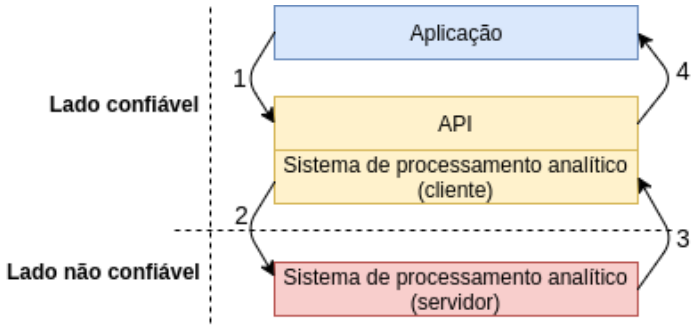


Figura 7: Visão lógica das camadas de um sistemas de processamento analítico seguro

oferecerem uma interface que facilita o uso aos clientes, estes sistemas fazem otimizações sobre as interrogações dos clientes. Geralmente, estes sistemas de processamento seguro seguem uma arquitetura composta por duas entidades, a entidade confiável e a não confiável. A entidade confiável, corresponde ao ponto de acesso dos clientes, enquanto que a entidade não confiável é onde a maioria do processamento é realizado, sendo este serviço controlado por terceiros. Desta forma, os dados passam a ser controlados pelos fornecedores deste serviço, sendo necessário assegurar a sua segurança.

Originalmente, foram criados sistemas baseado em bases de dados *SQL* para resolver essas limitações de segurança, sendo o *CryptDB*[48] pioneiro. Para o paradigma *Not Only SQL (NoSQL)*, também foram desenvolvidos sistemas como o *SafeNoSQL*[39] que, tal como o *CryptDB*, permitem que a computação seja realizada numa infraestrutura não confiável.

Na figura 7 é apresentada uma visão lógica que os sistemas de processamento analítico seguro seguem. Do lado confiável temos a aplicação que necessita resultados de processamento analítico. Esta comunica através de uma *API* com o sistema de processamento seguro que cifra as interrogações a enviar para o servidor(1). Após a receção da interrogação(2), o servidor procede ao processamento e envia os resultados para o cliente(3). Do lado não confiável existe a componente de armazenamento onde estão guardados os dados, podendo ser uma base de dados, um sistemas de ficheiros, entre outros. Para além desta, existe também uma componente de processamento analítico que vai ser responsável pela maioria do processamento. Caso não seja possível processar de forma segura a informação, esta é enviada para o lado confiável para ser decifrada e processada.

2.3.1 *Monomi*

Os sistemas originais de processamento seguro como o *CryptDB*, embora introduzissem o processamento seguro de uma grande conjunto de interrogações *SQL*, eram incapazes de processar eficientemente interrogações mais complexas. Desta forma, de modo a possibilitar a execução eficiente de interrogações *SQL* surgiu o *Monomi*[57]. Este sistema faz um pré-

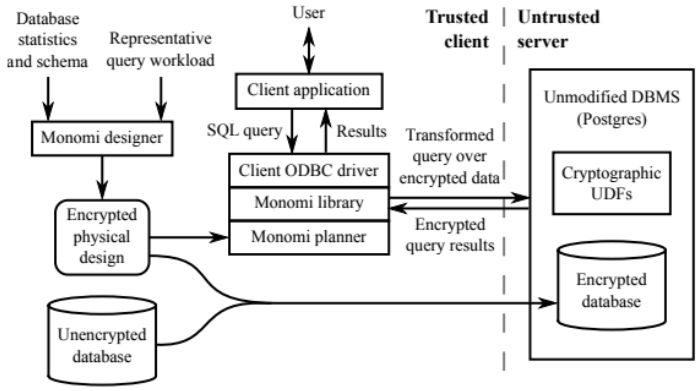


Figura 8: Arquitetura do sistema *Monomi*[3]

processamento das interrogações do utilizador e divide-as entre o cliente (confiável) e o servidor(não confiável), de forma a que a sua execução seja o mais eficiente possível(modelo de execução *split client-server*). Na figura 8 é apresentada a arquitetura do *Monomi*.

Contudo, existem vários desafios inerentes à construção deste sistema. Primeiro, interrogações sobre grandes volumes de dados são limitadas pelo *I/O* do sistema, visto que os esquemas criptográficos vão aumentar significativamente o tamanho dos dados, prejudicando o desempenho do sistema. Para além disto, a complexidade das interrogações faz com que a computação seja ineficiente ou impossível sobre o dados cifrados.

Para resolver o primeiro desafio, o *Monomi* segue o modelo de execução *split client/server*, que permite que a computação seja dividida entre o cliente e o servidor. Este sistema é baseado no *CryptDB*[48], um sistema que oferece confiabilidade sobre sistemas que utilizam bases de dados relacionais. São introduzidas várias técnicas que melhoram o desempenho para interrogações específicas, tais como, pré-processamento por linhas, técnicas criptográficas eficientes quanto ao espaço, adições homomórficas e pré-filtragem. Para resolver o último desafio são introduzidos dois módulos, o *designer* e o *planner*. O primeiro é responsável pela construção de um esquema físico otimizado, sendo este processo efetuado na configuração inicial do sistema. O *designer* recebe o esquema físico da base de dados e uma carga de trabalho representativa das interrogações que o sistema vai processar. Quanto ao *planner*, é responsável por determinar a melhor divisão da execução das interrogações, construindo um plano otimizado de execução, sendo que cada interrogação feita pelo utilizador passa por esta componente.

Para proteger a informação, o *Monomi* recorre a esquemas criptográficos, *Standard Encryption*, *Deterministic Encryption*, *Order-Preserving Encryption*, *Paillier Encryption*, *Searchable Encryption*, e ainda de forma a poupar espaço, o *Monomi* recorre a esquemas de *Format-Preserving Encryption*.

3 Retirado do artigo *"Processing analytical queries over encrypted data"*[57]

Em comparação com o sistema sobre o qual é construído, o *Monomi* garante o mesmo nível de segurança, mas permite que sejam feitas mais interrogações. Em termos de desempenho, o *Monomi* apresenta melhorias devido às otimizações e ao seu modelo de execução.

2.3.2 *Seabed*

O *Seabed*[46], construído sobre o *Apache Spark*, surgiu para colmatar os problemas que os esquemas criptográficos usados por sistemas anteriormente propostos traziam (*Monomi* e *CryptDB*), permitindo também processamento seguro dos dados. Os esquemas criptográficos utilizados por sistemas precedentes tem custo de computação elevado (por exemplo, *order-preserving encryption*), tornando-os inviáveis para aplicações de processamento em tempo real. Para além disso, esquemas determinísticos são vulneráveis a ataques de inferência[42].

De forma a resolver esta vulnerabilidade e trazer otimizações em comparação com os sistemas anteriores, o *Seabed* introduz dois novos esquemas criptográficos, o *Additively Symmetric Homomorphic Encryption (ASHE)* e o *SPLayed ASHE (SPLASHE)*. O *ASHE* é um esquema homomórfico três ordens de grandeza mais rápido que *Paillier Encryption*, contudo, este esquema não cumpre um requisito típico dos esquemas homomórficos. Este requisito obriga a que o tamanho de um criptograma não aumente com o número de operações feitas sobre este. Visto que o *ASHE* não respeita esse requisito, os criptogramas vão ocupar mais espaço que necessário, comparativamente a *Paillier*. A criação do esquema *SPLASHE* foi motivada pela vulnerabilidade que os esquemas determinísticos apresentam face a ataques por inferência, visto que o uso de esquema determinísticos revelam a frequência dos valores de forma a permitir validação de igualdades. Este esquema tem dois modos de funcionamento, o básico e o avançado. No modo básico, o esquema opera da seguinte forma: dada uma coluna C e um número de valores discretos d que esta pode assumir são criadas d colunas que substituem a coluna C, sendo cada coluna denominada C_d. Quando o valor da coluna C na linha n é x então o valor da coluna C_x na linha n é fixado a 1, por exemplo, se C contêm duas ocorrências de x então a coluna C_x vai conter dois valores fixados a 1 nas linhas em que x ocorre em C. Os restantes valores da coluna C_x são fixados a 0. Se as colunas resultantes do uso deste modo de funcionamento, forem cifradas usando *ASHE* os criptogramas são aleatórios para o atacante, tornando assim o sistema seguro contra ataques por inferência. Contudo, este modo de funcionamento é impraticável quando a gama de valores de d é muito grande, visto que este modo aumenta o consumo de armazenamento num fator de d. O modo avançado tenta resolver este problema apenas criando novas colunas para os valores mais frequentes. Para os valores que ocorrem menos que um número determinado de vezes é criada uma coluna auxiliar que será responsável pelo seu armazenamento. Contudo, este modo traz várias limitações: necessidade de conhecer a interrogação ou a distribuição dos dados antes da sua execução, dificuldade em gerir

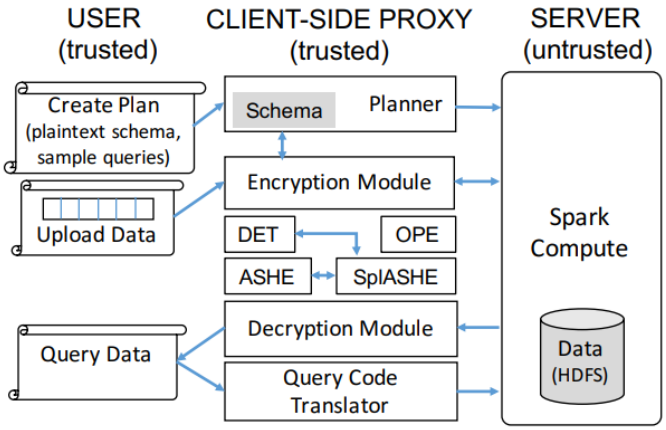


Figura 9: Arquitetura do sistema *Seabed*[4]

dados em que a distribuição varie rapidamente e os custos adicionais de armazenamento. Ambos os modos de funcionamento do *SPLASHE* requerem mais espaço de armazenamento, sendo que o uso do modo avançado em aplicações realistas requer 10 vezes mais espaço de armazenamento.

A arquitetura do *Seabed* apresentada na figura 9 recorre a uma componente de planeamento (*planner*, tal como o *Monomi*), para determinar os esquemas criptográficos a usar em cada coluna. Esta componente necessita de um esquema da base de dados e de uma carga de trabalho representativa das interrogações que o sistema vai processar. Para o armazenamento de dados estes são enviados para o módulo criptográfico do *Seabed* para serem cifrados com a cifra correta, que foi decidida no *planner*, sendo carregados para a componente de armazenamento. A componente de tradução de interrogações(*Query Code Translator*) é responsável por transformar as interrogações de forma a que sejam eficientemente executadas sobre a informação cifrada no lado não confiável. O servidor, após processar a interrogação, envia o resultado para o módulo encarregue de decifrar os dados, sendo feito o resto do processamento caso necessário.

2.3.3 *Opaque*

Ao contrário dos sistemas anteriores, o *Opaque*[61] recorre a *hardware* seguro de forma a proteger e efetuar a computação. O *hardware* seguro usado pelo *Opaque*, *Intel SGX*[26], consiste num conjunto de operações que permitem que o código dos utilizadores reserve regiões privadas de memória, denominadas de enclaves, contudo, isto obriga a que as nós suportem *Intel SGX*. Este sistema, construído sobre o *Spark SQL*, introduz soluções para colmatar o problema de análise de padrões que os enclaves de *hardware* sofrem. Esta vulnerabilidade pode ser explorada por um sistema operativo malicioso que é capaz de

4 Retirado do artigo *"Big data analytics over encrypted datasets with seabed"*[46]

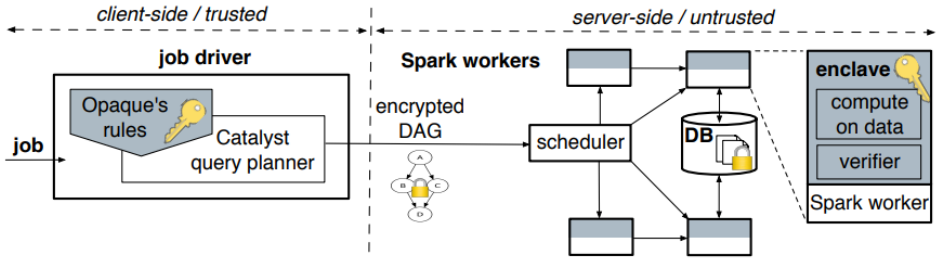


Figura 10: Arquitetura do sistema *Opaque*[5]

inferir informação sobre os dados cifrados através da monitorização do acesso à memória por parte da aplicação. Este sistema também introduz um mecanismo de verificação da computação de forma detetar se um atacante corrompe o resultado do processamento.

Na arquitetura apresentada na figura 10, temos apenas uma componente do lado confiável, o *Catalyst Query Planner*, sendo responsável pela construção do plano de execução da interrogação. Esta é a única alteração em relação à arquitetura do *Spark SQL*, sendo que o *Opaque* move esta componente para o lado confiável para evitar que um fornecedor malicioso do serviço de nuvem controle o planeador, resultando em execução de interrogações incorretas. O *scheduler*, responsável por gerir as tarefas entre os nós *Spark*, é colocado do lado não confiável, visto que há mecanismos para verificar se a computação foi corrompida. Esta componente também é responsável pela tolerância a falhas, por exemplo, caso um dos nós esteja lento ou tenha deixado de responder, o *scheduler* pode atribuir a tarefa desse nó a outro nó.

O *Opaque* tem três modos de funcionamento, o *encryption mode*, *oblivious mode* e o *oblivious pad mode*. Com a utilização do *encryption mode* o *Opaque* oferece garantias de confidencialidade e integridade, não protegendo o sistema de ataques que monitorizem os acessos à memória. O *oblivious mode*, para além de oferecer as mesmas garantias que o *encryption mode*, adicionalmente protege o sistema de ataques baseados em padrões de acesso aos dados. De forma a proteger o sistema contra ataque baseados em fugas do tamanho dos dados é utilizado o *oblivious pad mode*, assegurando as garantias dos outros modos de funcionamento. Ao recorrer a enclaves, a computação segura é efetuada no lado não confiável.

Segundo a arquitetura descrita na figura 10, uma interrogação assume o seguinte fluxo: inicialmente é gerado um *DAG* no *query planner*, que é cifrado e enviado para o lado não confiável (servidor). O *scheduler* trata então de distribuir o trabalho pelos nós *Spark*. Por fim, o resultado é enviado para lado confiável onde é decifrado.

De modo a garantir a privacidade e integridade dos dados, o *Opaque* usa *AES* no modo de operação *GCM*[35]. Este modo combina o modo *CTR*, explicitado anteriormente, com o

5 Retirado do artigo *"Opaque: An oblivious and encrypted distributed analytics platform"*[61]

modo de autenticação *Galois*, o que traz confidencialidade e verificação da integridade dos dados.

Em termos de desempenho, o *Opaque*, utilizando o *encrypted mode*, apresenta resultados que variam entre 58% de aumento de desempenho a 2.5x de perda de desempenho quando comparado com o *Spark SQL*, utilizando o *Big Data Benchmark*[8] para realizar a comparação do desempenho dos sistemas. Os ganhos no desempenho devem-se ao facto do *Opaque* utilizar *C++* no enclave, enquanto que o *Spark SQL* sofre perdas devido ao uso da *Java Virtual Machine (JVM)*. No modo *oblivious* o desempenho do *Opaque* é bastante penalizado, mais precisamente em 1.6-62x em relação ao *Spark SQL*, o que se deve ao aumento de segurança do sistema, protegendo-o da análise de padrões de acesso à memória e rede.

2.3.4 *PrivApprox*

Numa outra abordagem, o *PrivApprox*[50] oferece processamento em tempo real, garantindo simultaneamente a privacidade da informação. Ao contrário dos sistemas descritos anteriores, o *PrivApprox* usa o conceito de computação aproximada. Ao invés de processar completamente um conjunto de dados, é feito *sampling*, trocando a precisão dos resultados pela velocidade de processamento, o que é importante visto ser um sistema de processamento em tempo real. Para além disto, o *PrivApprox* não usa esquemas criptográficos tradicionais para proteger os dados, ao invés disto, recorre a *XOR-based encryption*. Se dois clientes querem trocar uma mensagem de tamanho l, começam por trocar uma chave privada M_k de tamanho l. De seguida, para cifrar a mensagem, o emissor aplica a operação de *XOR* à mensagem, obtendo o criptograma ($M_c = M \oplus M_k$). Depois de transmitido o criptograma (M_c), o recetor vai aplicar a chave trocada anteriormente a esse criptograma para obter a mensagem original ($M = M_c \oplus M_k$). Este esquema é extremamente eficiente quando comparado com os esquemas anteriores, contudo, após a troca de várias mensagens é possível descobrir a chave privada usada. Na figura 11 é apresentada a arquitetura do *PrivApprox*.

O sistema compreende quatro componentes, o investigador, o cliente, os *proxies* e o *aggregator*. O cliente guarda os dados localmente dos seus dispositivos e subscreve interrogações do sistema, sendo que estas são criadas pelos investigadores. Os *proxies* são responsáveis por transmitir as interrogações entre os clientes e receber as suas respostas. Estas respostas são processadas no *aggregator*, que descodifica as mensagens, calcula a estimação de erro e retorna o resultado ao analista. Concluindo, o *PrivApprox* é um sistema de processamento em tempo real de dados descentralizados que oferece privacidade aos clientes, contudo, se um grande número de clientes for malicioso, os resultados podem ser comprometidos.

6 Retirado do artigo *"Privapprox: Privacy-preserving stream analytics"*[50]

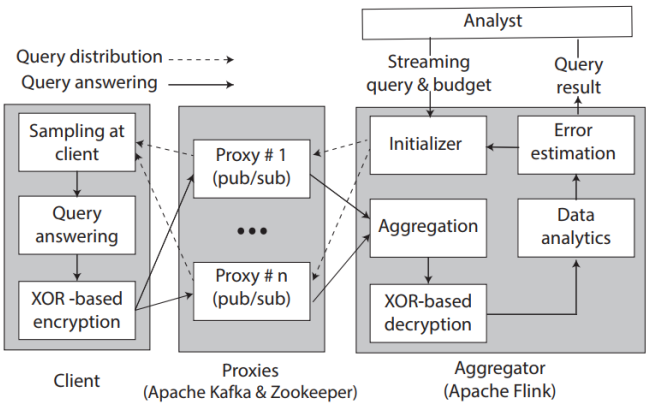


Figura 11: Arquitetura do sistema *PrivApprox*[6]

2.4 DISCUSSÃO

Tal como foi discutido anteriormente, as soluções de processamento analítico seguro fornecem um conjunto limitado de operações para o processamento dos dados. Além disso, algumas dessas soluções sugerem abordagens com um impacto muito elevado no desempenho do sistema analítico e aumentam o volume de dados a armazenar com o compromisso de dar suporte a um maior número de operações. Para além disso, estas soluções apresentam fraca modularidade e flexibilidade na escolha de diferentes técnicas que permitem suportar aplicações com diferentes compromissos em termos de segurança, desempenho e funcionalidade.

3

ARQUITETURA

De forma a colmatar as falhas apresentadas anteriormente, esta dissertação propõe uma nova plataforma denominada *SafeAnalytics*, um sistema de processamento analítico modular e flexível que permite o processamento dos dados de forma segura. Este sistema é configurável de forma a maximizar o seu desempenho, garantindo que o processamento é feito sem revelar informação sensível dos dados processados. Ainda, denominado de *SafeAnalytics*, permite que os dados sejam processados garantindo a privacidade destes, com o melhor desempenho possível. Visto que são usadas várias técnicas criptográficas, que permitem operações sobre os dados cifrados, é necessário ter um compromisso configurável entre a segurança, funcionalidade e o desempenho. Para além disso, de forma a diminuir os recursos necessários para processar os dados de forma segura, é seguido o modelo *split-execution* que permite partir uma interrogação em partes, permitindo a sua execução numa infraestrutura confiável e não confiável, seguindo o conceito utilizado no *Monomi*.

3.1 SAFEANALYTICS

O *SafeAnalytics* garante a privacidade do processamento dos dados de forma transparente. Tal como alguns dos sistemas enunciados em capítulos anteriores, o *SafeAnalytics* segue um modelo em que o processamento é dividido entre uma parte confiável e uma não confiável.

Na figura 12 é apresentada a arquitetura, composta por duas componentes: uma presente no lado confiável que corresponde a uma plataforma de computação em que o cliente tem confiança, por exemplo, uma infraestrutura privada gerida por este, e outra no lado não confiável que corresponde a uma plataforma de computação em que o cliente não tem confiança, por exemplo, serviços de nuvem.

Cada uma destas componentes possui um motor de processamento, responsável pelo processamento dos dados e um módulo criptográfico responsável pela segurança dos dados e do processamento feito sobre estes.

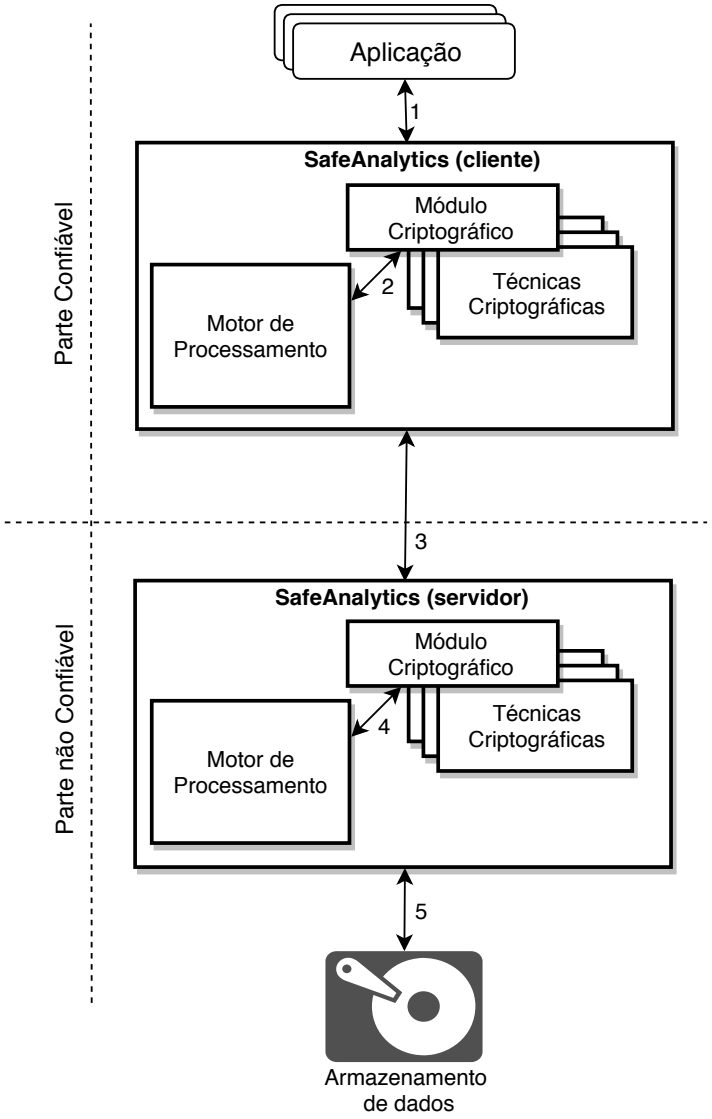


Figura 12: Arquitetura do sistema *SafeAnalytics*

3.1.1 *Motor de processamento e Módulo criptográfico*

Tal como referido anteriormente, a arquitetura é dividida em duas partes o que permite ao utilizador diminuir os recursos utilizados da sua infraestrutura durante o processamento. Ambas as partes do sistema são constituídas por dois módulos, o motor de processamento e módulo criptográfico. Na parte confiável do sistema, o motor de processamento é inicialmente responsável pelo planeamento do processamento analítico, utilizando o módulo criptográfico presente na parte confiável para preparar as interrogações. Por exemplo, uma interrogação que realiza filtros de igualdade é executada, assim, é necessário cifrar os campos correspondentes utilizando técnicas apropriadas, neste caso determinística.

O comportamento das componentes da parte não confiável do sistema é semelhante. Na parte não confiável, o motor de processamento é responsável por realizar parte da interrogação, seguindo o planeamento realizado previamente, utilizando o modo criptográfico para operações que não sejam possíveis de executar diretamente sobre os dados, por exemplo, agregações. O restante processamento é concluído na parte confiável do *SafeAnalytics*, após o módulo criptográfico decifrar os dados.

A possibilidade da adição de novas técnicas criptográficas aos módulos torna esta arquitetura modular, permitindo que o sistema evolua com a descoberta de novas técnicas que permitem novas operações sobre dados cifrados.

3.1.2 *Fluxo do processamento na plataforma*

De forma a facilitar a compreensão do funcionamento do *SafeAnalytics* é apresentado o comportamento de uma aplicação que executa uma interrogação que contem filtros de igualdade para realizar processamento analítico. O processo é iniciado quando a aplicação envia a interrogação para o *SafeAnalytics* (1), sendo intercetada pelo motor de processamento presente na parte confiável do sistema. A interrogação é processada, otimizada e modificada pelo módulo criptográfico (2) de acordo com as técnicas criptográficas que utiliza, sendo que neste exemplo, como são utilizados filtros de igualdade, os campos são cifrados com uma cifra determinística. Após realizadas as alterações necessárias, é criado um plano de execução da interrogação.

O processamento continua na parte não confiável (3), em que o motor de processamento recebe a informação sobre o processamento que necessita de realizar, recorrendo ao módulo criptográfico (4) caso não seja possível executar algumas operações mais complexas, por exemplo, agregações. De seguida os dados armazenados (cifrados) são lidos (5), executando diretamente sobre estes operações que o sistema permita, neste caso filtros de igualdade.

Após o término do processamento na parte não confiável, os resultados processados são enviados para a parte confiável (3) onde são decifrados pelo módulo criptográfico (2).

Dependendo das interrogações, o processamento pode não ser completamente concluído na parte não confiável do sistema, obrigando a que seja concluído pelo motor de processamento confiável. Por fim, os resultados são enviados para a aplicação (1). Concluindo, a parte não confiável do sistema não tem acesso aos dados decifrados, e, dependendo das técnicas criptográficas utilizadas, não consegue tirar informação do processamento realizado.

4

IMPLEMENTAÇÃO DO SAFEANALYTICS

A arquitetura para a plataforma de processamento analítico seguro, apresentada no capítulo 3, contempla o processamento de dados de forma seguro. Neste capítulo será feita uma descrição detalhada da implementação do *SafeAnalytics*, descrevendo as decisões tomadas, as funcionalidades implementadas e algumas contribuições para projetos utilizados no âmbito dissertação.

4.1 APACHE SPARK

Para a implementação do *SafeAnalytics* foi utilizado o *Apache Spark* como plataforma do processamento analítico visto tratar-se de uma plataforma *open-source* amplamente utilizada. Para além de ser o maior projeto *open-source* na área de processamento de dados[10], o *Apache Spark* tem bibliotecas para processamento de grafos, *machine learning*, processamento de *streams* e processamento estruturado de dados.

A implementação do *SafeAnalytics* focou-se no módulo de processamento estruturado, o *Spark SQL*, que para além de oferecer os benefícios do processamento relacional, por exemplo, interrogações declarativas aos utilizadores, permite o uso de interrogações analíticas complexas, por exemplo, *machine learning*. A criação deste módulo do *Spark* foi inspirada pelo *Shark[59]*, um sistema que combina interrogações relacionais com interrogações analíticas complexas oferecendo tolerância a falhas. Este sistema otimiza o processamento através de técnicas que permitem a otimização dinâmica das interrogações em *run-time* através de estruturas de dados apropriadas (*DAGs*) e técnicas de armazenamento, por exemplo, armazenamento orientado à coluna.

A principal diferença entre os modelos de computação do *Spark SQL* e o *Spark Core* é a plataforma relacional para interrogar e persistir dados estruturados utilizando interrogações relacionais que podem ser expressas em *SQL* e em *Domain Specific Language (DSL)* que o *Spark SQL* oferece.

O *Spark SQL* introduz uma abstração de dados tabular denominada *DataFrame* criada para trabalhar com dados estruturados e semi-estruturados. Esta estrutura inspirada nos *RDDs* adquire algumas das suas características, como a imutabilidade, computação em memória,

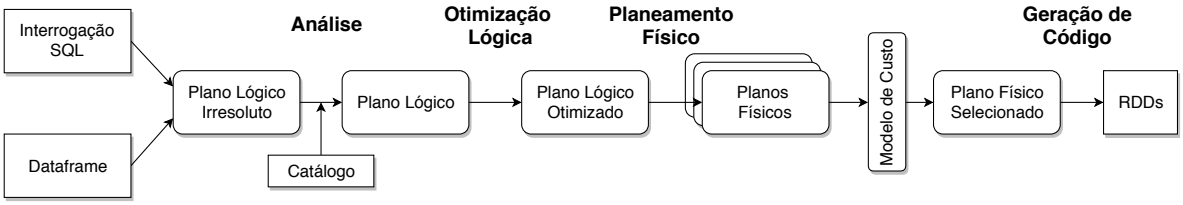


Figura 13: Diferentes fases da execução de uma interrogação

resiliência, distribuição e paralelização, diferindo destas estruturas na utilização de um esquema para representar o tipo de dados, sendo comparável a uma tabela de uma base de dados relacional com a adição de otimizações.

Tirando partido das características da linguagem de programação utilizada na conceção do *Spark*, nomeadamente o *pattern matching* e *quasi quotes* do *Scala*, foi construída uma componente extensível, denominada de *Catalyst optimizer*, que permite a otimização de interrogações. A extensibilidade desta componente possibilita a adição de novas otimizações e características ao *Spark SQL*, bem como adicionar novas regras ou tipos de dados ao *Catalyst*.

O *Catalyst* possui diferentes bibliotecas para otimização das interrogações relacionais e um conjuntos de regras para lidar com as diferentes fases da execução das interrogações: análise, otimização lógica, plano físico, e geração de código, apresentadas na figura 13. A primeira fase começa com uma *Abstract Syntax Tree (AST)* devolvida pelo *SQL parser* ou quando é uma computação é lançada sobre um *Dataframe* criado utilizando a *API* que este fornece. Esta fase tem como propósito criar um plano lógico com dependências irresolutas, ou seja, verificar se atributos ou relações coincidem com tabelas conhecidas. Após a criação deste esquema lógico são aplicada várias regras com o intuito de resolver tipos de dados e atributos sem mapeamento.

A fase de otimização lógica consiste na aplicação de regras ao plano lógico, entre as quais, *constant folding*, *predicate pushdown*, *projection pruning*, *null propagation*, entre outros. Além disto, é possível aplicar otimizações *cost-based*, que se resumem à criação de múltiplos planos utilizando diferentes regras e calculando o seu custo.

O próximo passo na otimização passa por converter o plano lógico produzida na etapa anterior num plano físico, utilizando operadores compatíveis com o *Spark*. Tal como a fase anterior, são usadas otimizações baseadas em custo e regras de forma a determinar o melhor plano de execução.

Por fim, na última fase de otimização da interrogação, geração de *Java bytecode*, consiste em transformar a árvore que representa a expressão *SQL* em uma *AST*. Esta estrutura é então passada ao compilador de *Scala* gerando o *bytecode*, que por sua vez será distribuído pelos *executors* para efetuarem o processamento da interrogação.

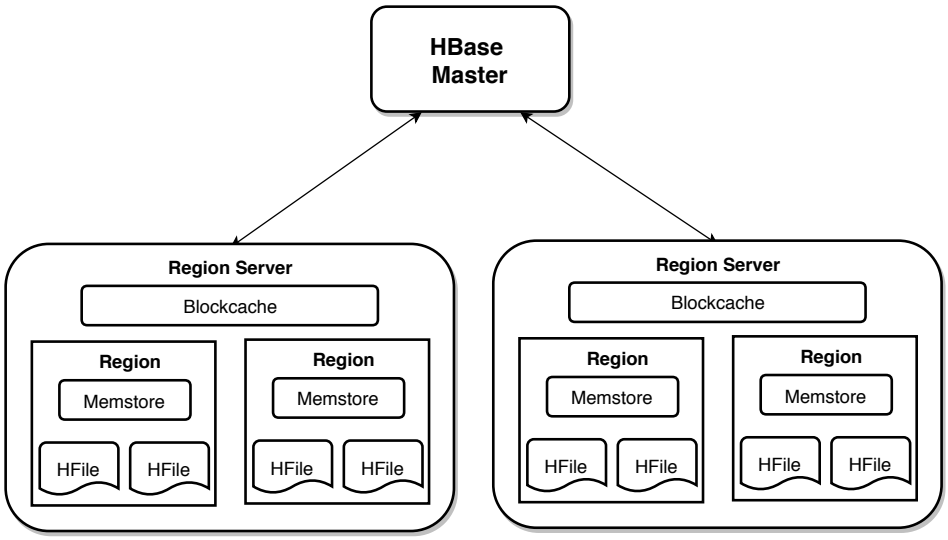


Figura 14: Arquitetura do sistema HBase

Apesar de uma solução baseada em *Apache Spark* e *HDFS* permitir que alguma computação seja feita diretamente sobre os dados, por exemplo filtragem sobre colunas, não permite que tal seja feito ao mesmo tempo que oferece proteção sobre dados, ou seja, não existe um mecanismo que permita o processamento diretamente sobre os dados cifrados. Para além disso, muitas aplicações necessitam de utilizar bases de dados para armazenar e permitir pesquisas sobre estes. Assim é necessário utilizar uma base de dados que permita tanto armazenar a informação, como realizar processamento sobre estes. Desta forma, foi escolhido o *HBase*, uma base de dados não relacional, apropriada para o contexto de processamento analítico. Adicionalmente, existe uma solução implementada sobre esta base de dados, o *SafeNoSQL*, que garante a privacidade dos dados, bem como a realização de processamento seguro obre estes.

4.2 HBase

O *HBase*[5] é uma base de dados distribuída não relacional criada para armazenar e processar grandes volumes de dados. A sua arquitetura, representada na figura 14, segue um modelo *master/slave*.

A componente responsável pela coordenação do sistema é o *HMaster*. Este é responsável pela administração do sistema, balanceamento de carga das vários *Region Servers*, recuperação de *Region Servers* em caso de falhas, entre outras tarefas. Os *Region Servers* recebem pedidos de leitura e escrita do cliente e atribuem essa tarefa à região que contêm a respetiva informação. Um *Region Server* pode manter até 1000 regiões, e, cada um possui uma *blockcache* com o propósito de diminuir a latência nas leituras de dados. Quanto aos

FuncionarioID	Dados Pessoais		Dados Profissionais	
	Nome	Idade	Designação	Salário
1	Pedro	25	Engenheiro	15 000
2	João	29	Doutor	25 000

Tabela 2: Exemplo de uma tabela *HBase*

mecanismo de *caching*, o *HBase* utiliza a *blockcache* ao nível do *Region Server* e *memstore* ao nível das regiões. A *blockcache* é o mecanismo principal para garantir a baixa latência de leituras aleatórias. Esta funciona da seguinte forma: quando é realizada a leitura de um bloco do *HDFS* este é escrito para a *blockcache*. Assim, leituras que utilizem frequentemente dados presentes nesse bloco, terão um desempenho melhorado. O espaço alocado para este mecanismo é, por padrão, 40% do tamanho total da *heap*. Por outro lado, a *memstore* tem como função armazenar escritas para *Region Servers* em memória. De forma a melhorar o desempenho de escritas, a *memstore* acumula dados até preencher o espaço que lhe foi alocado, sendo estes escritos para um *HFile* em disco. Assim, a *memstore* tem dois fins: aumentar o tamanho dos dados escrita para disco numa só operação, e reter dados escritos recentemente, para leituras rápidas subsequentes. É utilizado 40% do tamanho total da *heap* para a *memstore*.

As tabelas, compostas por uma *row key*, *column families* e *column qualifiers*, são particionadas horizontalmente em regiões, que contêm todas as linhas de uma tabela entre uma *start* e *end key*. Habitualmente, os dados correspondentes a uma *column family* são armazenados no mesmo *HFile*, podendo existir mais do que um, dependendo do tamanho dos dados. Na tabela 2 é apresentado um exemplo de uma tabela *HBase* que compreende uma *row key*, nomeadamente *FuncionarioID*, duas *column families*, Dados Pessoais e Dados Profissionais, subdivididas em dois *column qualifiers*, Nome e Idade, e Designação e Salário.

A *API* que o *HBase* disponibiliza para manipular os dados das tabelas é a seguinte:

- *Put*: Inserção de dados nas tabelas, em que é necessário indicar a *Row Key*, *Column Family* e *Column Qualifiers* e o valor a inserir.
- *Get*: Pesquisa de uma linha com uma determinada *Row Key*.
- *Scan*: Pesquisa de todas as linhas associadas a um conjunto contíguo de *Row Keys*.
- *Delete*: Remover registos com uma determinada *Row Key*.

Para além destas operações, o *HBase* suporta a utilização de filtros, permitindo reduzir o volume de dados a ser processado pelo cliente. Assim, é possível retornar um subconjunto de resultados, sem que este tenha que aplicar os filtros na sua infraestrutura. Dos filtros existentes, é dado destaque aos mais importantes no âmbito desta dissertação:

- *SingleColumnValueFilter:* Utilizado para filtrar dados baseando-se no valor das colunas.
- *RowFilter:* Utilizado para filtrar dados por *row keys*.
- *FilterList:* Utilizado para agrupar filtros.

4.3 SPARK HBASE CONNECTOR

Para utilizar o *HBase* seguro como sistema de armazenamento foi necessário recorrer a uma biblioteca externa que permita a comunicação entre os dois sistemas *Spark* e *HBase*. Apesar do repositório do *HBase* possuir um projeto para esse propósito, este tem muitas limitações. Assim, recorreu-se ao uso de um repositório externo para este propósito[37]. O *Spark-HBase Connector (SHC)* suporta a utilização do *HBase* como fonte de dados externa pelo *Spark* ao nível de *Dataframes*, sendo esta umas das vantagens de usar este repositório em comparação com o nativo que apenas opera a nível de *RDDs*. Um requisito na utilização do conetor é a necessidade da criação um esquema em formato *JSON*, denominado de catálogo, que mapeia tipo de dados entre a uma tabela *HBase* e um *Dataframe*. Com o suporte de *Dataframes* esta sistema consegue tirar partido das técnicas de otimização que o *Spark* oferece, como *data locality*, *predicate pushdown*, entre outros.

Data locality

Quando é necessário ler dados do *HBase*, o *Spark* vai co-alocar um *executor* com os dados para eliminar de transferência de dados pela rede.

Predicate pushdown

O *HBase* aplica filtros diretamente sobre dados, reduzindo a quantidade de dados lida. Assim, é possível integrar o *Spark SQL* com o *HBase* ao nível de *Dataframes* usufruindo das otimizações que estes oferecem.

4.4 SAFENOSQL

De forma a garantir a privacidade e segurança de dados num ambiente de computação em nuvem há duas possibilidades: utilização de sistemas que protejam os dados e os sirvam para serem processados numa plataforma segura, também conhecido como *encryption at rest*, não permitindo qualquer tipo de processamento na nuvem, ou utilização de sistemas que permitam o seu processamento seguro, mesmo que parcial, na nuvem.

Sistemas que utilizam *encryption at rest* para proteger os dados, como por exemplo o *Ranger KMS*, utilizado pelo *HDFS*, cifra ficheiros com técnicas que não possibilitam que

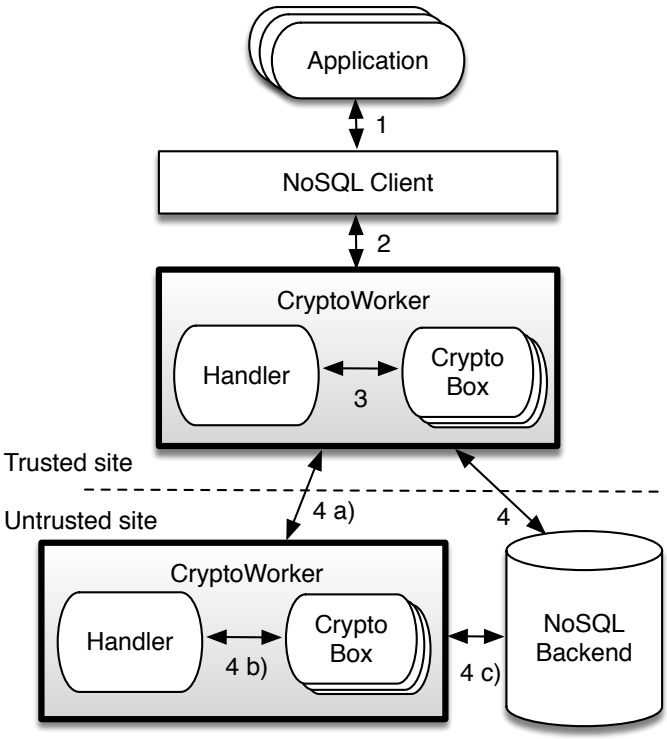


Figura 15: Arquitetura do SafeNoSQL[1]

estes sejam processados no provedor do serviço visto tratar-se de *end-to-end encryption*, ou seja, os dados são cifrados e decifrados apenas pelo cliente. Outra opção passa por usar sistemas que protejam os dados e ao mesmo tempo permitam que uma parte da computação seja realizada nestes. Para que tal seja possível é necessário que estes sistemas utilizem técnicas que permitam o processamento de dados cifrados sem que exista uma penalização de desempenho que torne a primeira opção mais viável. Assim, foi necessário selecionar um sistema que possuísse as características referidas anteriormente e que se ajustasse ao contexto de processamento de grandes volumes de dados. Posto isto, foi determinado o uso do *SafeNoSQL* como sistema de armazenamento e processamento seguro de dados.

O *SafeNoSQL* é um sistema modular e extensível que oferece de forma transparente privacidade e segurança a bases de dados *NoSQL*, implementado sobre *HBase*. A arquitetura deste sistema, apresentada na figura 15, é constituída por duas partes, uma parte segura em que está presente o cliente do sistema e uma parte não confiável que representa os prestadores de serviços de armazenamento na nuvem.

Esta plataforma implementada sobre *Apache HBase* utiliza módulos criptográficos, denominados de *CryptoBoxes*, para dar suporte a múltiplas técnicas criptográficas para proteger informação sensível. Isto torna o sistema extensível visto possibilitar a implementação

1 Retirado do artigo *"A Practical Framework for Privacy-Preserving NoSQL Databases"*[39]

FuncionarioID	Dados Pessoais		Dados Profissionais	
	Nome	Idade	Desginação	Salário
DET	STD	OPE	DET	OPE

Tabela 3: Técnicas criptográficas utilizadas para proteger uma tabela *HBase*

de *CryptoBoxes* com diferentes técnicas criptográficas que permitem diferentes tipos de operações sobre os dados. De momento, o *SafeNoSQL* suporta as seguintes *CryptoBoxes*:

- *Standard Encryption*: Utilizada para proteção de informação extremamente sensível, não suportando processamento direto sobre os dados.
- *Deterministic Encryption*: Usada em dados que necessitem de operações de igualdade.
- *Order-Preserving Encryption*: Permite filtros de ordem e igualdade sobre os dados.
- *Format-Preserving Encryption*: Para além de preservar o formato do conteúdo, permite operações de igualdade, tal como *Deterministic Encryption*.

Em suma, o *SafeNoSQL* oferece diferentes níveis de segurança dependendo do tipo de operações a que os dados estão sujeitos, assim, através de um ficheiro de configuração é possível definir as técnicas a usadas sobre os dados armazenados, fornecendo uma troca entre desempenho e segurança configurável pelo utilizador do sistema.

De forma a demonstrar como o *SafeNoSQL* protege a informação armazenada e permite que seja realizado processamento sobre esta, é apresentada uma tabela 3 com os esquemas criptográficos utilizados para proteger a informação apresentada na tabela 2. A metodologia seguida para a escolha das técnicas a utilizar baseia-se na carga de trabalho a que a tabela vai ser sujeita, por exemplo, neste caso é necessário efetuar pesquisas pela designação do funcionário, bem como, utilizar filtros de ordem sobre a idade e sobre o salário. Assim, de forma a manter o melhor compromisso entre a segurança e operações a realizar sobre a informação segura, todos os campos que não são utilizados diretamente no processamento são protegidos com *Standard Encryption*. Para possibilitar operações de igualdade, os *column qualifiers FuncionarioID* e *Designação* são protegidos com *Deterministic Encryption*. Por fim, de forma a possibilitar operações de ordem, sobre os *column qualifiers Idade* e *Salário*, a informação é protegida com *Order-Preserving Encryption*.

4.5 SAFEANALYTICS

O *SafeAnalytics* é resultado da integração do *Apache Spark*, uma plataforma de processamento analítico, e o *SafeNoSQL*, um sistema de armazenamento e processamento seguro implementado sobre *HBase*. Durante a realização desta integração foi necessário implementar

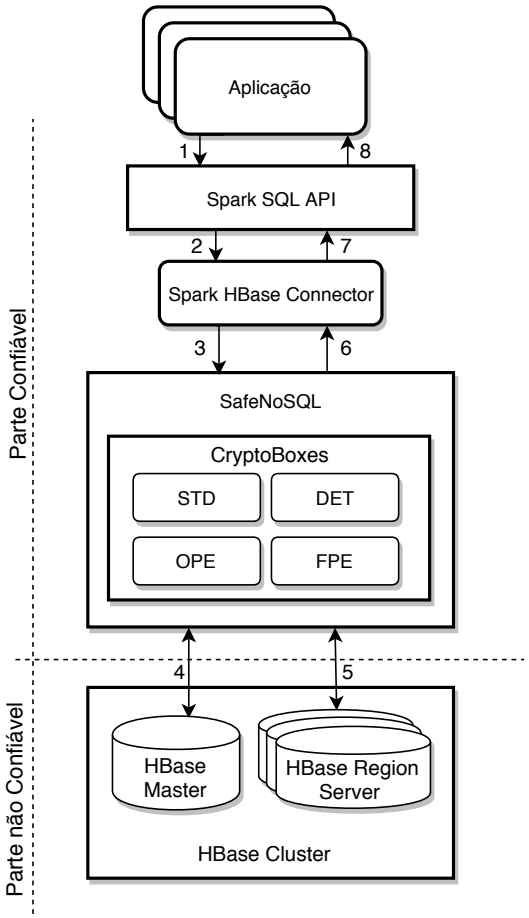


Figura 16: Implementação do sistema *SafeAnalytics*

e alterar algumas funcionalidades de forma a que possível construir uma plataforma de processamento analítico seguro. Tal como foi referido na secção 4.4, o *SafeNoSQL* utiliza *Cryptoboxes* que suportam *Standard Encryption, Deterministic Encryption, Order-Preserving Encryption* e *Format-Preserving Encryption* para garantir a segurança e privacidade dos dados, bem como o processamento sobre estes.

Em termos de alterações necessárias aos sistemas utilizados, foram feitas alterações mínimas ao *SafeNoSQL*, visto que usava uma versão mais antiga do *HBase* e esta não era suportada pelo conetor. Assim, utilizou-se a versão 1.2.6 do *HBase* e a versão 2.2.0 do *Apache Spark*. Quanto ao *Spark HBase Connector* foi criada uma cópia do projeto original, que suporta o *HBase* sem garantias de segurança e, foi criado com base neste, um novo projeto que suporta o *SafeNoSQL*.

Assim, foi criado um protótipo do *SafeAnalytics* representado na figura 16. De modo a explicitar o comportamento do sistema, é apresentado o fluxo de uma interrogação no sistema. O processo é iniciado quando uma aplicação faz uma interrogação ao sistema (1), esta é transformada e otimizada pelo *Spark* (2) que vai utilizar o conetor para fazer a

leitura dos valores necessários (3). De seguida, a interrogação é intercetada pelo *SafeNoSQL*, nomeadamente a componente presente na parte segura do sistema, que vai cifrar valores presentes na interrogação de acordo com o esquema de técnicas a utilizar sobre os dados. O sistema determina em que *Region Servers* estão os dados requeridos (4) e realiza a leitura destes, aplicando os filtros diretamente sobre os dados (o *predicate pushdown* referido anteriormente) (5). Depois de realizada a leitura os dados são enviados para parte segura do sistema, onde são decifrados pelas *Cryptoboxes* respetivas (6) e realizado o restante processamento (6)(7). Por fim, os resultados são enviados para a aplicação (8).

Durante a integração do *Spark HBase Connector* no *SafeAnalytics* foram encontrados vários problemas que prejudicavam a desempenho do sistema, bem como a exatidão dos resultados. Um dos problemas iniciais baseava-se na escolha e estaticidade das chaves. Visto o conetor apenas suporta chaves compostas de forma simplista, em que era necessário que os $N - 1$ campos pertencentes à chave composta tivessem tamanho fixo, onde N é o número total de campos da chave composta, e visto que várias tabelas não cumpriam este requisito, foi criada uma alternativa em que é gerada uma coluna em que os elementos desta não se repetem. Esta solução baseia-se no incremento de um número inteiro até ao número de linhas da tabela, sendo que levanta uma pequena penalização em termos de espaço de armazenamento utilizado.

Quanto à estaticidade das chaves, foi necessário criar uma estrutura, nomeadamente um dicionário, no qual são guardados os nomes das tabelas e o respetivo número de linhas. Assim, ao invés de utilizar as chaves estáticas presentes no conetor, o que criava regiões sem dados visto que nenhuma das chaves se encaixava no intervalo de chaves pré-definido, foram utilizadas as chaves definidas no dicionário. Com o uso de um ficheiro de configuração em que era indicado o nome e o número das tabelas era possível tornar esta solução ainda mais genérica, não obrigando o utilizador a modificar a estrutura diretamente no código.

Outro problema encontrado, que influenciava diretamente a exatidão dos resultados, passava pela forma como alguns filtros eram tratados. Em causa estão os filtros de $\geq$ e de $\leq$ serem processados da mesma forma que os filtros de $>$ e de $<$. De forma a explicitar o comportamento do conetor, são apresentadas duas interrogações distintas.

$$SELECT\ c\_customer\_first\_name\ FROM\ customer\ WHERE\ c\_birth\_year\ \geq\ 1990$$

$$SELECT\ c\_customer\_first\_name\ FROM\ customer\ WHERE\ c\_birth\_year\ >\ 1990$$

Ao contrário do que é esperado, as duas interrogações acima seguem o mesmo comportamento. Na função responsável pelo processamento de filtros, os filtros de $\geq$ e de $\leq$ são convertidos em $>$ e de $<$, respetivamente. Assim, foi necessário adicionar dois novos casos

de forma a suporta de forma correta os filtros, para que as interrogações devolvessem os resultados esperados.

5

AVALIAÇÃO EXPERIMENTAL

Após a definição da arquitetura e a implementação de um protótipo da plataforma *SafeAnalytics* é necessário efetuar testes de desempenho para validar o seu impacto com a integração dos sistemas *SafeNoSQL* e *Apache Spark*. Este capítulo tem como objetivo a comparação do desempenho de soluções que permitem processamento analítico seguro dos dados, em que a plataforma *SafeAnalytics* é constituída pelo *Apache Spark* e pelo *SafeNoSQL*, com sistemas que permitem processamento analítico sem garantir a segurança dos dados, nomeadamente *Apache Spark* utilizando *HBase*. Assim, recorrendo a cargas de trabalho realistas, os sistemas são comparados para medir a penalização no desempenho que a adição de segurança acarreta.

5.1 TPC-DS

O *TPC-DS* é uma plataforma de avaliação de sistemas de suporte à decisão que modela os vários aspetos geralmente aplicáveis a um sistema deste tipo de forma realista. Esta plataforma representa sistemas de suporte à decisão que:

- Examinam grandes volumes de dados;
- Respondem a questões de negócio aplicáveis no quotidiano;
- Executam interrogações com diferentes requerimentos operacionais e complexidades;
- Sistemas caracterizados por alto consumo de *CPU* ou *IO*;
- Sistemas para soluções *"Big Data"*, por exemplo, baseados no ecossistema *Hadoop*.

A carga de trabalho do sistema modela uma cadeia de fornecedores de produtos que os vende e distribui quer por lojas físicas, como por lojas virtuais. O sistema mantém informação de negócio relevante, tal como dados de clientes, dados dos produtos, ordens de produtos, entre outros. A plataforma modela um sistemas de suporte à decisão em que os utilizadores convertem os dados armazenados em *business intelligence*. Assim, a carga

de trabalho pode representar qualquer indústria que transforme dados operacionais em *business intelligence*.

De forma a abordar os diversos tipos de interrogações e comportamentos do utilizadores encontrados num sistema de decisão ao suporte, o *TPC-DS* utiliza um modelo de interrogações generalizado. Este modelo permite que a plataforma de avaliação capture aspetos importantes de interrogações *OLAP* de natureza iterativa e interativa bem como interrogações mais complexas de *data mining* e interrogações com comportamento conhecido denominadas de interrogações de consulta.

De modo a caracterizar as interrogações criadas pelos utilizadores o *TPC-DS* define quatro classes de interrogações:

- *Reporting Queries* - Estas interrogações são executadas periodicamente para responder a questões pré-definidas sobre a saúde operacional e financeira do negócio. Neste grupo de interrogações o utilizador geralmente define parâmetros da interrogação, por exemplo, um intervalo temporal, localização ou marca de um produto.

- *Ad-hoc Queries* - Estas interrogações são construídas para responder a questões específicas do negócio. A principal diferença entre estas interrogações e as interrogações da classe anterior é o conhecimento prévio que o administrador da base de dados possui quando define o esquema da base de dados, ou seja, o esquema é construído de forma a otimizar o desempenho de interrogações da classe *reporting*, que são bem conhecidas aquando da implementação do esquema de base de dados.

- *Iterative OLAP Queries* - Este tipo de interrogações permitem a exploração e a análise dos dados para a descoberta de relações e tendências novas.

- *Data Mining Queries* - Estas interrogações processam grandes volumes de dados de forma a aprender o comportamento e modas futuras dos clientes, permitindo ao negócio tomar decisões com base nesse conhecimento. Esta classe de interrogações consiste geralmente em agregações e extração de grandes volumes de dados.

5.1.1 *Implementação*

Das várias implementações consideradas foi selecionado o projeto *open-source* da *IBM*[14] visto permitir a adição de novas funcionalidades. Com esta plataforma é possível avaliar e medir a desempenho do *Spark SQL* num ambiente realista, suportando um fator de escala até 100TB. Contudo esta plataforma apenas suporta a avaliação do *Spark SQL* sobre o sistema de ficheiros local, ou seja, obriga a que os dados sejam movidos manualmente para o *HDFS* para o testar numa configuração distribuída.

5.1.2 *Contribuições*

De forma a dotar a plataforma de avaliação com capacidade para avaliar o *SafeAnalytics*, foi feita uma extensão ao sistema da *IBM* para realizar as contribuições necessárias[1]. Assim, foram feitas duas contribuições para a plataforma.

Primeiro, a plataforma não permite de forma transparente a utilização do *HDFS* como sistema de armazenamento de dados, assim, de modo a facilitar a avaliação do *Apache Spark* numa configuração distribuída utilizando o *HDFS* foi implementado um módulo para permitir a sua utilização sem a necessidade de mover dados manualmente.

Dado que foi utilizado o *HBase* como sistema de armazenamento de dados do *SafeAnalytics*, foi necessário implementar um módulo para permitir a sua utilização na plataforma de avaliação. Tal como foi referido anteriormente, uma limitação do conetor obrigada à adição de uma coluna aos dados gerados pelo *TPC-DS*. Posto isto, como segunda contribuição à plataforma de avaliação, foi criada uma função para modificar os dados antes de serem carregados para o *HDFS*. Tal como na solução anterior, é utilizado um ficheiro de configuração para definir em que nós são guardados dados e feito o processamento.

5.1.3 *Análise das interrogações*

Devido ao elevado número de interrogações que a plataforma de avaliação[56] possui foram escolhidas 8 representativas do total de 99 interrogações. Assim, foram escolhidas duas interrogações de cada classe referida anteriormente (nomeadamente *Reporting*, *Ad-hoc*, *Iterative* e *Data Mining*), usando a especificação da plataforma[41] e alguns estudos que foram feitos sobre a mesma[47] como suporte.

De seguida, é feita uma descrição detalhada de cada interrogação, sendo especificado em que infraestrutura do sistema de processamento, confiável ou não confiável, são realizadas. Na medida em que as técnicas criptográficas utilizadas não suportam todo o tipo de operações (por exemplo, agregações), há a necessidade de processar alguns dados na infraestrutura confiável.

Interrogação 24

Esta interrogação é composta por duas iterações relacionadas entre si. Na primeira iteração é calculado o valor total de produtos de uma determinada cor nas vendas de loja num mercado específico, organizados por nome do cliente e nome da loja. Estes valores calculados dizem respeito a clientes que não vivem no país do seu nascimento nem na proximidade da loja, e apenas são listados os clientes cujos totais sejam maiores do que 5% do valor médio.

1 Disponível no *GitHub*: https://github.com/RumpleZ/spark-tpc-ds-performance-test

Na segunda iteração faz exatamente a mesma interrogação variando apenas os parâmetros desta.

Ambas as iterações são constituídas por 4 operações de agregação, nomeadamente 3 somatórios e 1 média. Ambos os tipos de agregação são executados pelo *Spark*, após a leitura dos dados, na infraestrutura segura do sistema. Quanto aos filtros executados diretamente no *HBase*, temos dois filtro de igualdade, um sobre a cor dos produtos e outro sobre o mercado em questão.

Interrogação 27

Para todos os itens vendidos em lojas localizadas em seis estados durante um dado ano, encontrar a quantidade média de produtos, uma lista de preços médios, uma lista de médias de preços de venda, média da quantidade de cupões para um dados sexo, estado civil, educação e dados demográficos do cliente.

O *Spark* vai ser responsável pela execução das 4 agregações encontradas nesta interrogação, nomeadamente médias, na infraestrutura segura do sistema. Nesta interrogação existem 4 filtros de igualdade executados na infraestrutura não segura, nomeadamente sobre o ano, sexo, estado civil e educação dos clientes.

Interrogação 31

Lista de regiões onde a percentagem de crescimento nas vendas pela Internet é consistentemente maior quando comparada com a percentagem de crescimento nas vendas de loja nos três primeiros trimestres de um dado ano.

Na infraestrutura não confiável do sistema apenas é executado 1 filtro de igualdade referente ao ano em causa na interrogação. Quanto às operações relevantes executadas no lado confiável, temos 2 agregações, nomeadamente somatórios.

Interrogação 40

Computar o impacto da mudança do preço de um produto nas vendas, calculando o total de vendas para produtos num período de 30 dias antes e depois da mudança de preço. Os produtos são agrupados pela localização do armazém do qual foram entregues.

Para além de realizar o filtro de igualdade de ano na infraestrutura não confiável, também são executados 2 filtros de ordem correspondentes ao preço do produto em questão. As operações de agregação realizadas na infraestrutura confiável do sistema são 2 somatórios.

Interrogação 70

Calcular o *ranking* de lucro líquido de vendas por estado e região dado um ano e determinar os 5 estados mais rentáveis.

Esta interrogação possuí dois tipos de agregações diferentes, somatórios e *ranks*. Visto tratar-se de uma interrogação de cálculo de um *ranking*, existem 5 operações desse tipo. O outro tipo de agregação é um somatório, utilizado em 3 ocasiões. Ambas as agregações são realizadas do lado confiável do sistema. Do lado não confiável é aplicado 1 filtro de ordem sobre um período de tempo.

Interrogação 73

Calcular o número de clientes com potenciais de compra específicos e que em três anos consecutivos fizeram compras em lojas localizadas em 4 regiões, e que compraram de 1 a 5 produtos numa só compra. Apenas as compras dos dois primeiros dias do mês são consideradas.

Esta interrogação contém 2 filtros de igualdade, ambos aplicados ao potencial de compra, e 3 filtros de ordem aplicados sobre os dias das compras. Estes são aplicados diretamente sobre os dados cifrados na infraestrutura não confiável. Em relação às agregações presentes, existem dois tipos, *ranking* e contagem, aparecendo cada uma desta operações 1 e 4 vezes, respetivamente. Estas agregações são aplicadas sobre os dados no lado seguro do sistema.

Interrogação 81

Encontrar os clientes e a sua informação pessoal que retornaram produtos comprados do catálogo mais do que 20% da média de retornos para um dado estado num dado período de tempo.

Existem 2 filtros de igualdade referentes ao estado e ao ano em questão na interrogação, ambos executados diretamente sobre o *HBase* na infraestrutura não confiável do sistema. Existem dois tipos de agregações, ambas executadas na infraestrutura confiável do sistema, nomeadamente um somatório e o cálculo de uma média.

Interrogação 82

Encontrar os clientes que tendem a gastar mais dinheiro em lojas virtuais do que em lojas físicas.

Ao contrário do resto das interrogações selecionadas, esta não possui operações de agregação. Assim, a maior parte da computação é realizada na infraestrutura não confiável, onde são efetuados 6 filtros de ordem, executados sobre preços, inventário disponível e datas.

A classificação das interrogações, segundo os classes definidas anteriormente é apresentada na tabela 4. Para além desta classificação, a classe *Ad-hoc* é subdividida em duas classes que representam os recursos mais utilizados, sendo que a interrogação 82 é *IO-Intensive* e a interrogação 70 é *CPU-Intensive*.

Classe	Iterative		Data Mining		Reporting		Ad-hoc	
Interrogação	24	31	40	82	27	73	81	70

Tabela 4: Classificação das interrogações

5.1.4 *Esquema da base de dados*

Para proteger os dados e ao mesmo tempo permitir que sejam feitas operações sobre estes, foi necessário analisar as interrogações selecionadas e ver o tipo de operações que cada uma realiza, como apresentado na secção anterior. Assim, de acordo com a análise realizada, foi criado um esquema de base de dados que maximiza a quantidade de operações realizadas na infraestrutura não confiável do sistema.

O esquema, presente em anexo nas tabelas 16, 17 e 18, apresenta algumas particularidades. Primeiro, todas tabelas possuem um campo em texto limpo referente à chave. A razão disto deve-se a uma limitação do conetor que não permite que o campo referente à chave das tabelas seja cifrado. Contudo, isto não levanta nenhuma limitação a nível da segurança oferecida visto se trata de uma coluna gerada aquando a inserção dos dados na base de dados, que consiste num inteiro auto incremental, tornando os valores da coluna únicos, o que é um requisito para ser a chave da tabela.

De forma a elevar as garantias de segurança dos dados decidiu-se proteger todas colunas que não são alvo de processamento direto do lado não confiável com *Standard Encryption*, assim, um atacante não consegue retirar qualquer informação dessas colunas.

As restantes colunas são cifradas com *Deterministic Encryption* e *Order-Preserving Encryption*, dependendo das operações que necessitam de suportar.

5.2 METODOLOGIA

Para avaliarmos o *SafeAnalytics* foram criados vários ambientes de teste diferentes. O primeiro foco foi verificar as diferenças de desempenho entre dois ambientes, em que no primeiro os dados são co-alocados com as componentes responsáveis pelo processamento, e no segundo as componentes de execução e os dados encontram-se em componentes distintas. Esta comparação serve principalmente para analisar o compromisso entre explorar a localidade de dados e utilizar mais recursos visto que as componentes de armazenamento e de processamento estão instaladas em nós diferentes.

Assim é possível simular dois cenários realistas em que os utilizadores necessitam de processamento analítico seguro, sendo que cada um oferece diferentes garantias de segurança. O primeiro protótipo proposto oferece garantias de segurança mais baixas, contudo garante um melhor desempenho e custos de infraestrutura mais baixos.

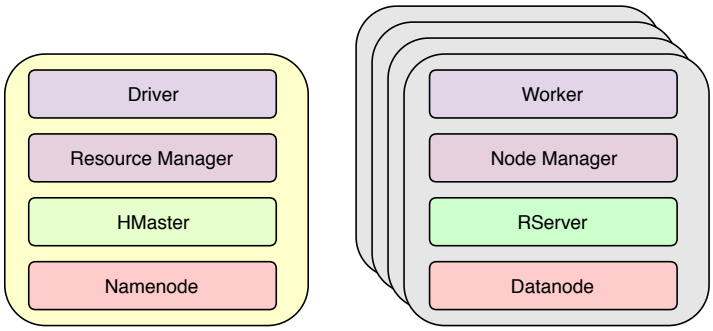


Figura 17: *Cluster* com co-alocação de dados com a plataforma de processamento

Neste cenário, apresentado na figura 17, os *Region Servers* são co-alocados com os *executors* do *Spark*, o que apesar de dar melhores garantias de desempenho, obriga a que os dados em memória estejam decifrados. Ou seja, aquando de uma interrogação, é feito o processamento sobre os dados protegidos pelo *SafeNoSQL* e são lidos para memória onde são decifrados e é realizado o resto do processamento.

Foi construído um *cluster* composto por cinco nós, em que distribuição de processos foi feita da seguinte forma: o *HBase Master* corria num nó e os *HBase RegionServers* nos restantes nós, sendo atribuídos 2GB de *heap* para cada um. O *cluster Spark* também foi distribuído pelas mesmas cinco nós, sendo que o *Spark Driver* foi co-alocado com o *HBase Master* e os *executors* nos restantes nós, isto para aproveitar a co-alocação dos dados.

De forma a aumentar as garantias de segurança oferecidas propõe-se recorrer a outro *cluster*, onde o sistema de armazenamento de dados e a plataforma de processamento analítico são desacoplados, apresentado na figura 18.

Este *cluster* requer que sejam utilizados mais recursos e, visto que desacopla o processamento confiável do processamento não confiável, causa um aumento na transferência de dados transferidos pela rede, prejudicando o desempenho do sistema. Por outro lado, a utilização do modelo de *split-execution* permite que a computação seja dividida em duas partes, sendo que uma parte é processada numa infraestrutura não confiável, por exemplo, um serviço de computação em nuvem e a outra, processada na infraestrutura privada do utilizador.

De forma a testar o modelo de *split-execution*, dividindo o sistema em duas partes, confiável e não confiável, foi construído sobre um *cluster* composto por dez nós. O *HBase* seguro foi distribuído por cinco nós seguindo o mesmo processo de *setup* anterior, em que o *HBase Master* foi colocado num nó e os *Region Servers* nós restantes quatro nós. As configurações em relação ao *setup* anterior foram mantidas, em que foi atribuído 2GB de *heap* para cada um.

Quanto ao *cluster* de *Spark*, também foi distribuído por cinco nós, uma para o *Spark Driver* e as restantes para os *executors*. Em termos de recursos utilizados, decidiu-se manter as configurações da secção anterior para uma melhor comparação, ou seja, ficaram recursos

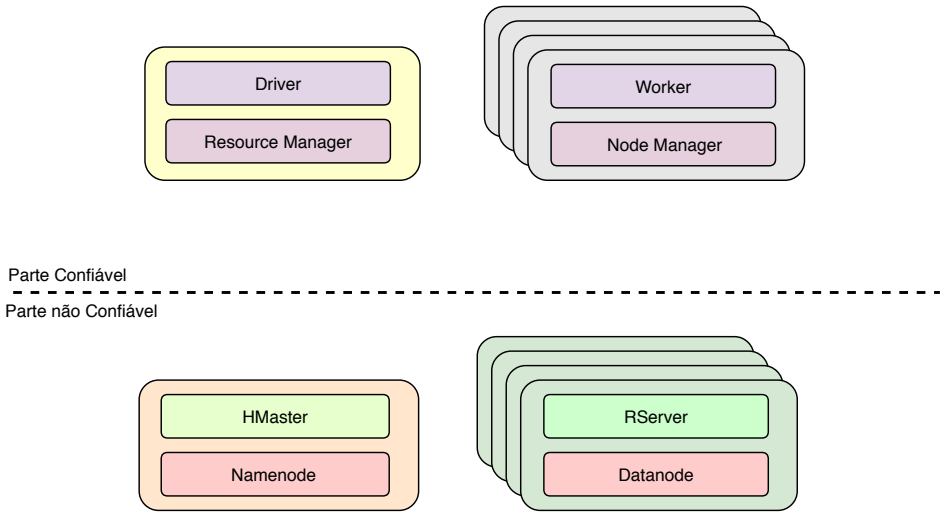


Figura 18: *Cluster* seguindo o modelo *split-execution*

por utilizar em prol da verificação do impacto de desacoplar os dados da plataforma de processamento. Esta divisão de recursos é apresentada na tabela 5. Assim, tal como no *setup* anterior, cada *executor* do *Spark* utiliza 1 *core* e 2GB de memória.

O *cluster Spark* foi distribuído por cinco nós em que uma foi alocada para o *Spark Driver* e mais quatro para os *executors*. Ao contrário do *cluster* da fase 1 o *Spark* e o *HBase* não ficam co-alocados pelas razões descritas anteriormente. O *cluster HBase* também foi distribuído por cinco nós: um nó alocada para o *HBase Master* e quatro alocadas para os *Region Servers*.

A distribuição de recursos pelos vários processos seguiu um processo semelhante ao da fase anterior, como é apresentado na tabela 5.

De forma a otimizar o processamento foi feito um pequeno estudo para verificar quais as configurações ideais para o *cluster*. Dos cinco nós, quatro são utilizados para processamento, enquanto que um é utilizado para submissão de trabalhos e gestão de recursos. Cada um dos quatro nós possuem um processador de quatro *cores* e 8GB de memória. Um *core* de cada nó é reservado para o sistema operativo e outros processos essenciais, sendo a mesma ideia é aplicada a 1GB de memória. Assim sobram três *cores* e 7GB de memória por nó. A divisão destes recursos para os restantes foi feita tendo em conta a explicação dada no capítulo do estado da arte na secção do *YARN* e do *Spark*, em que se chegou à conclusão que a solução ótima é um meio termo entre número de *executors* e os recursos que cada um destes possui. Assim sendo, foi decidida a seguinte divisão dos recursos: dos restantes três *cores* um é alocado para os *daemons* do *Hadoop* e do *HBase* enquanto que os restantes dois *cores* são reservados para os *executors* do *Spark*; dos restantes 7GB de memória 3GB

	S.O.	Spark	Hadoop & HBase	Total
Core	1	2	1	4
Memória	1	4	3	8

Tabela 5: Divisão dos recursos pelos diferentes processos do *cluster*

reservados para o *HBase* e para os *daemons Hadoop*, sendo os restantes 4GB divididos entre dois *executors* do *Spark*.

Assim, foram criadas três variantes de um sistema de processamento analítico com o propósito de ser feita uma comparação entre o desempenho, garantias de segurança e recursos utilizados por cada uma:

- *Spark* e *HDFS*: Tem como propósito a comparação do desempenho do sistema, no caso de utilizar um sistemas de ficheiros e de uma base de dados como forma de armazenar os dados
- *Spark* e *HBase*: Serve como *baseline* para a comparação do sistema sem garantias de segurança
- *Spark* e *SafeNoSQL*: Sistema de processamento analítico com garantias de segurança

De forma a avaliar as três variantes implementadas procedeu-se à execução das interrogações, sendo cada uma executada três vezes para calcular a média e o desvio padrão.

Posto isto, foi gerada uma carga de trabalho de 10GB pelo *TPC-DS* para avaliar ambos os *clusters*. Utilizando o esquema de base de dados referida na secção 5.1.4 o sistema foi pré-populado.

5.2.1 *Configuração do ambiente de testes*

A avaliação foi feita num *cluster* composto por cinco nós com duas especificações de *hardware* diferentes. O primeiro grupo de nós é equipado com um processador *Intel i3-3240* com quatro *cores* de 3.4GHz, 8GB de memória *DDR3* de 1333MHz e um disco *Hard Disk Drive (HDD)* de 500GB com interface *SATA III*. O segundo grupo de nós é equipado com um processador *Intel i3-2100* com quatro *cores* de 3.1GHz, 8GB de memória *DDR3* de 1333MHz e um disco *HDD* de 250GB com interface *SATA II*. A ligação de rede entre nós é efetuada através de um *switch* com interface *Gigabit*. De modo a facilitar a distinção entre os tipos de nós ao longo deste capítulo, nós com a interface *SATA II* são denominados nós de primeira geração, enquanto que nós com *SATA III* são denominadas nós de segunda geração.

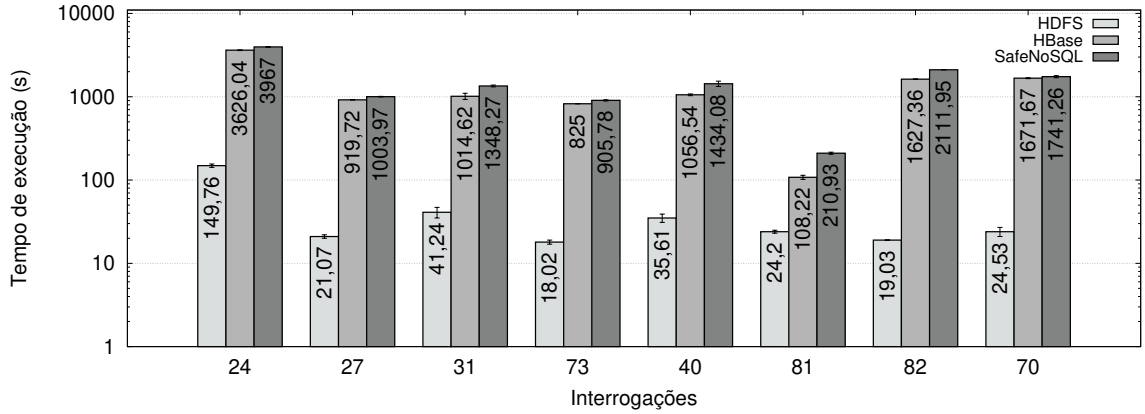


Figura 19: Tempo de execução das interrogações

5.3 AVALIAÇÃO EXPERIMENTAL

O impacto na fase de carregamento dos dados para o *HBase* sem garantias de segurança é de, aproximadamente, 4 vezes o tamanho dos dados originais, ou seja, 40GB e o tempo de carregamento dos dados ronda as 8 horas. Para o *SafeAnalytics* o tamanho aumentou para cerca de 50GB com um tempo de carregamento semelhante.

5.3.1 *Avaliação do sistema com co-alocação dos dados com executors*

Na figura 19 são apresentados os resultados obtidos para a carga de trabalho do *TPC-DS* das três variantes. Tal como se pode verificar, em média o *SafeAnalytics* quando comparado com o *HBase* sem segurança, apresenta um custo adicional de 14,73%, tendo como pior caso a interrogação 81 em que a penalização é de 94,91%. Esta perda de desempenho deve-se à forma como o *HBase* armazena os dados, ou seja, o problema é a distribuição desbalanceada das tabelas pelos *Region Servers* não permitindo a leitura em paralelo das tabelas pelos *executors*. Na secção de discussão deste capítulo (5.4), é feita uma explicação mais detalhada deste problema.

Com o *cluster* composto por *Spark* e *HDFS* temos um sistema cerca de 39 e 47 vezes mais rápido comparativamente ao sistema utilizando *HBase* e *SafeNoSQL*, respetivamente. Esta diferença de desempenho deve-se principalmente à forma como o *Spark* armazena os dados no *HDFS*. A informação é comprimida utilizando o *Snappy*[17], um algoritmo de compressão que oferece velocidades de compressão e descompressão altas, garantindo um taxa de compressão razoável. Para além disto, a informação é armazenada utilizando um formato de armazenamento colunar, nomeadamente *Parquet*[16]. Em comparação com o *HBase*, este formato requer um menor de espaço de armazenamento devido à forma com codifica a

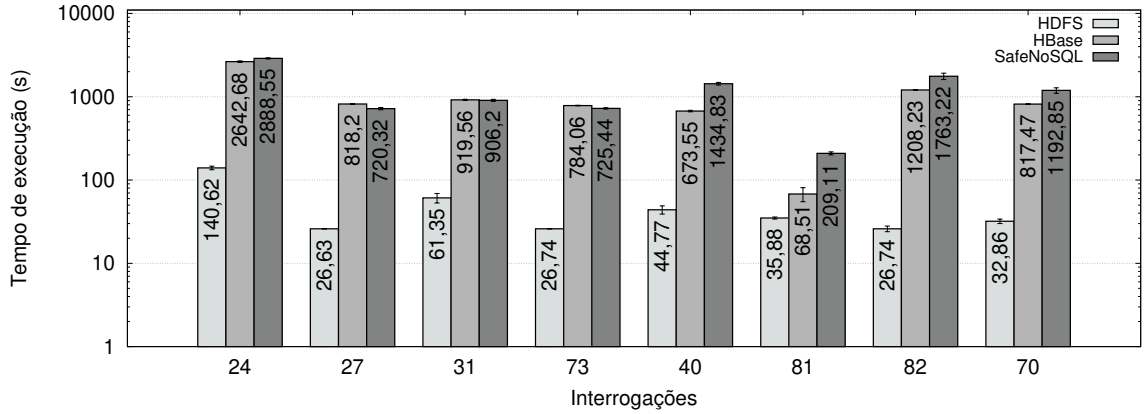


Figura 20: Tempo de execução das interrogações

informação. Para além disso, devido à forma como armazena informação, consegue ignorar informação que não é utilizada nas interrogações.

5.3.2 *Avaliação do sistema seguindo o modelo de split-execution*

A figura 20 apresenta os resultados obtidos com a execução das interrogações nas três variantes do sistema. Tal como no *setup* com a propriedade de co-alocação, o *SafeNoSQL* apresenta um desempenho muito semelhante ao *HBase*, sendo mais rápido em algumas interrogações. Assim, conclui-se que o sistema com garantias de segurança apresenta um custo adicional de 20,39% em termos de tempo de execução, sendo este valor correspondente à diferença percentual entre as médias do tempo de execução de ambos os casos. Das interrogações utilizadas, duas delas apresentam quebras de desempenho significativas, nomeadamente a interrogação 40 e 81. Ao contrário do esperado, existem duas interrogações que apresentam melhorias no tempo de execução em relação à sua versão sem segurança. Isto deve-se à forma como os dados foram armazenados nas bases de dados (segura e não segura), especificamente ao mau balanceamento dos dados pelos *Region Servers*. Em relação à variante do sistema que utiliza o *HDFS*, o sistema com garantias de segurança é, em média, 27 vezes mais lento, enquanto que o sistema sem garantias de segurança é 21 vezes mais lento.

Análise do consumo de recursos

Para além do tempo de execução das interrogações, foi também o medido o consumo médio dos recursos em todas os nós do *cluster* utilizando a ferramenta da monitorização *dstat*[11] no mesmo momento em que as interrogações foram executadas. A utilização dos recursos foi medida para todas as interrogações. Contudo, na medida em que os resultados recolhidos são semelhantes em todas as interrogações, iremos nesta secção detalhar sobre os

Interrogação 70	Spark					HBase				
	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10
CPU (%)	86.48	7.67	4.42	3.45	7.82	25.33	0.11	0.09	23.75	0.11
Memória (GB)	2.81	2.88	2.47	2.75	3.35	1.50	3.03	3.04	3.06	3.02
Disco - Leitura (KB/s)	≈ 0	≈ 0	≈ 0	≈ 0	≈ 0	0.02	≈ 0	≈ 0	44 592	≈ 0
Disco - Escrita (KB/s)	0.05	0.46	0.26	0.26	0.65	0.03	0.01	0.01	0.01	0.01
Rede - Receção (MB/s)	0.03	4.15	2.36	2.09	4.18	0.01	≈ 0	≈ 0	0.12	≈ 0
Rede - Envio (MB/s)	0.02	0.17	0.16	0.14	0.17	≈ 0	0.03	≈ 0	12.28	≈ 0

Tabela 6: Resultados do *dstat* para a interrogação 70

resultados para apenas uma interrogação, nomeadamente a interrogação 70, apresentado na tabela 6 a informação recolhida. Os restantes resultados são apresentados no Anexo A. Assim, utilizando a interrogação 70 como exemplo, podemos verificar que os nós 1 e 6, que representam o *Spark Driver* e o *HBase Master* respetivamente, fazem uso mais intensivo do *CPU*. Mais precisamente, o nó 1 é responsável pela criação de um plano de execução, delegação de tarefas aos *executors* e apresentação dos resultados obtidos, o que vai de encontro aos resultados observados. Os nós responsáveis pelo processamento dos dados, apresentam uma utilização de *CPU* abaixo do que seria de esperar visto tratar-se de uma interrogação que faz uso intensivo do *CPU* dada a categoria em que se enquadra. A explicação encontrada para este fenómeno deve-se ao tempo que o sistema passa em cada tarefa. Depois de feitos alguns testes concluiu-se que o sistema passa cerca de 90% do tempo de execução a fazer leituras dos dados. Assim, o *CPU* só é utilizado intensivamente nos restantes 10% do tempo de execução.

No que diz respeito aos resultados do *HBase* seguro, verificou-se que um dos *Region Servers* faz a leitura de quase todos os dados, sendo isto um indício de algum problema com o balanceador do *HBase*, visto que os dados não foram distribuídos de forma equilibrada pelos *Region Servers*.

Na secção seguinte são referidos problemas que foram encontrados na construção do *Safe-Analytics* e testes que foram realizados para verificar o impacto de alterações na configuração dos sistemas subjacentes.

5.4 Discussão

Nas secções anteriores foi feita a comparação entre as variantes quer no sistema com coalocação dos dados com os *executors*, quer no sistema que segue o modelo de *split-execution*. Assim, nesta secção é ser feita uma comparação entre os resultados obtidos entre os dois modelos utilizados. Na tabela 7 é apresentado o ganho percentual da utilização do modelo de *split-execution*, que, como seria de esperar, a variante do sistema que utiliza *HDFS* apresenta piores resultados com a utilização do modelo de *split-execution*. A razão

	Interrogações							
	24	**27**	**31**	**73**	**40**	**81**	**82**	**70**
HDFS	6,1	-29,39	-48,76	-48,39	-25,72	-48,28	-40,51	-33,96
HBase	27,12	11,04	19,22	4,96	36,25	36,69	25,76	51,1
SafeNoSQL	27,26	28,25	32,79	19,91	-0,05	0,86	16,51	31,5

Tabela 7: Diferença percentual no desempenho entre as duas configurações

de tal acontecer, passa pela eficiência com que os dados são lidos, ou seja, visto que são guardados num formato colunar e utilizam compressão. Assim, a coalocação dos *executors* com os dados é vantajoso, visto que o desempenho é limitado pela transferência de dados pela rede.

Por outro lado, as variantes que utilizam *HBase* e *SafeNoSQL* apresentam melhores resultados no cenário com *split-execution*. Isto deve-se principalmente à disponibilidade de recursos, ou seja, no modelo com coalocação, os nós utilizam *CPU* e disco na leitura dos dados, não sendo viável realizar o processamento nas mesmas, assim a transferência de dados para nós com os *executors* desacoplados dos dados não tem um impacto tão notório no desempenho.

Foram também realizados vários testes com configurações distintas de modo a aumentar o desempenho de *SafeAnalytics*, nomeadamente a implementação que oferece garantias de segurança e segue o modelo de *split-execution*. Visto que a razão da perda de desempenho do sistema é o *IO*, foram feitas algumas modificações ao nível do *HBase*, nomeadamente.

- Compressão dos dados
- Alteração do tamanho da *heap*
- Alteração do tamanho da *blockcache*
- Desabilitação da *blockcache*
- Balanceamento manual das regiões pelos *Region Servers*

A primeira alteração passou pela utilização de um algoritmo de compressão dos dados, utilizando o *Snappy* devido à sua elevada velocidade de compressão e descompressão e pela taxa de compressão razoável. Esta alteração também não apresentou melhorias em termos de desempenho, contudo diminuiu o espaço utilizado em cerca de 30%.

Outro teste passou pela alteração da *heap* e da *blockcache* do *HBase*. A *heap* do *HBase* é dividida em três partes: a *memstore* que utiliza, por omissão 40% do espaço de *heap* total, *blockcache* que também utiliza 40% da *heap* e os restantes 20% reservados para operações internas. Assim visto que no ambiente de *split-execution* existem recursos que não são utilizados no lado não confiável, alterou-se o tamanho da *heap* de 2GB para 5GB. Mais, visto

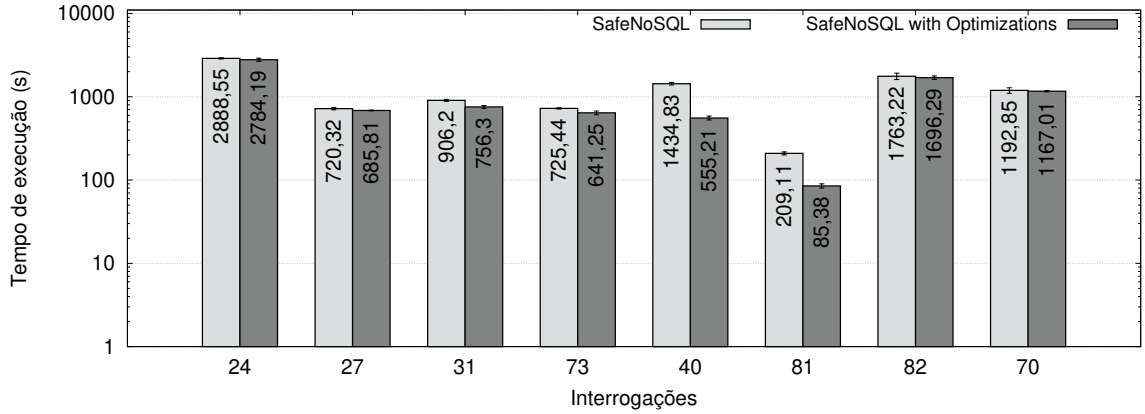


Figura 21: Tempo de execução das interrogações

que *memstore* é utilizada como mecanismo de *caching* de escritas para o *HBase*, modificou-se a porção de memória que utiliza para 5% (originalmente 40%). Assim, os restantes 75% da *heap* foram alocados para a *blockcache*, o mecanismo de *caching* para leituras, ou seja, 3.75GB. Das 8 interrogações executadas, apenas a interrogação 81 apresentou melhorias em relação ao sistema sem alterações, diminuindo o tempo de processamento em 30%. Como foi descrito na secção 5.1.3, esta interrogação não faz uso das tabelas de vendas, que se destacam das restantes pela quantidade de dados que contêm. Assim, visto que os dados utilizados nesta interrogação conseguem ser armazenados na *blockcache*, faz sentido o aumento de desempenho verificado. As restantes interrogações perdem desempenho visto que os dados não conseguem ser armazenados na *blockcache*, e as pausas para *garbage collection* aumentam visto que o tamanho da *heap* aumentou.

Assim, visto que o aumento da *heap* não apresenta melhorias para o sistema foi reduzida para o tamanho original (2GB). Além disto a *blockcache* foi desativada pelos mesmos motivos.

Por fim foi feita uma otimização para aumentar o paralelismo das leituras no sistema. Esta otimização consiste no balanceamento das regiões pelos diferentes *Region Servers*. Visto que existiam várias tabelas com as regiões com tamanho desequilibrado, e mal distribuídas foi feito o reparticionamento destas e foram movidas entre os *Region Servers* de forma a equilibrar o espaço utilizado por estes. As regiões também foram distribuídas pelos *Region Servers* para melhorar o paralelismo na leitura de dados.

De forma a verificar a viabilidade destas otimizações foram realizados novos testes, e comparados com a solução que seguia o modelo de *split-execution*, apresentados na figura 21.

Como se pode verificar, em todas as interrogações há melhorias no tempo de execução sendo que as otimizações aumentaram o desempenho em cerca de 20% em relação à versão sem estas.

6

CONCLUSÃO

Com a realização desta dissertação foi criado um sistema de processamento analítico seguro, o *SafeAnalytics*. Este sistema possibilita o armazenamento e o processamento de dados em plataformas de computação na nuvem oferecendo garantias de privacidade e segurança dados e o processamento seguro sobre estes. Após um estudo extenso do estado da arte atual dos sistemas de processamento analítico seguro, nomeadamente, a arquitetura, a implementação e as garantias de segurança e privacidade que oferecem, conclui-se que apesar de possibilitarem o processamento analítico seguro dos dados, a sua fraca modularidade e a sua área de aplicabilidade reduzida, penalizam o desempenho do sistema e reduzem a sua utilidade num contexto real.

Assim, com o conhecimento obtido, foi desenhado o sistema *SafeAnalytics*, que oferece uma arquitetura modular e extensível que colmata alguns dos problemas encontrados nos estado da arte atual de sistemas de processamento analítico seguro. Foi implementado um protótipo do *SafeAnalytics* sobre o *Apache Spark* e *SafeNoSQL*, que possibilita o processamento analítico seguro de forma transparente. Foram utilizados quatro módulos criptográficos que oferecem garantias de segurança distintas e possibilitam o processamento seguro de informação sensível.

Por fim, foi realizada a avaliação do sistema num conjunto alargado de *setups*, através da ferramenta de avaliação realista *TPC-DS*. Este sistema de avaliação simula uma carga típica de um sistema de venda de produtos de retalho. Em termos de desempenho o *SafeAnalytics* apresenta uma penalização de 20% em relação ao sistema base, composto pelo *Apache Spark* e o *HBase* que não apresenta garantias de confidencialidade.

Esta dissertação demonstra a importância de adotar soluções que contemplam diferentes técnicas criptográficas de forma a corresponder aos diferentes requisitos de desempenho, segurança e funcionalidade das diferentes aplicações. Baseando-se neste requisito, o trabalho consolidado nesta dissertação apresenta uma primeira proposta de plataforma para sistemas de processamento analítico seguro modulares, o que abre novas possibilidades em termos de trabalho futuro relacionado.

6.1 TRABALHO FUTURO

Para além da possibilidade da adição de novas técnicas criptográficas ao sistema para permitir que sejam feitas mais operações sobre dados cifrados, também seria interessante estudar novas formas de melhorar o desempenho, aumentar a quantidade de funcionalidades que suporta o processamento seguro, e ao mesmo tempo, fortalecer os modelos de segurança do *SafeAnalytics*. Por fim, um rumo interessante de seguir como trabalho futuro baseia-se na flexibilidade que o *SafeAnalytics* oferece. A metodologia abordada nesta dissertação utiliza o paradigma de *split-execution* para realizar o processamento seguro dos dados, em que a computação é dividida em dois domínios, confiável e não confiável. Esta abordagem, para além de ter o custos de transferências de dados, obriga a que o cliente possua poder de processamento. Assim, seria interessante integrar novas técnicas que permitam maximizar a quantidade de processamento realizado numa infraestrutura não confiável, recorrendo, por exemplo a tecnologias de *hardware* confiável, de modo a reduzir o poder de processamento necessário do lado do cliente.

BIBLIOGRAFIA

[1] Amazon kinesis – amazon web services (aws). URL https://aws.amazon.com/pt/kinesis/. Accessed on Dec. 17, 2017.

[2] Hdfs erasure coding, . URL https://hadoop.apache.org/docs/r3.0.0/hadoop-project-dist/hadoop-hdfs/HDFSErasureCoding.html.

[3] Welcome to apache flume, . URL https://flume.apache.org/. Accessed on Dec. 17, 2017.

[4] Apache hadoop yarn, . URL https://hadoop.apache.org/docs/r2.7.2/hadoop-yarn/hadoop-yarn-site/YARN.html. Accessed on Dec. 07, 2017.

[5] Apache hbase – apache hbase™ home, . URL https://hbase.apache.org/. Accessed on Dec. 17, 2017.

[6] Apache kafka, . URL https://kafka.apache.org/. Accessed on Dec. 17, 2017.

[7] Apache spark™ - lightning-fast cluster computing, . URL https://spark.apache.org/. Accessed on Oct. 27, 2017.

[8] URL https://amplab.cs.berkeley.edu/benchmark/. Accessed on Apr. 28, 2018.

[9] You are viewing this page in an unauthorized frame window. URL https://csrc.nist.gov/Projects/Block-Cipher-Techniques/BCM. Accessed on Dec. 17, 2017.

[10] Apache Spark - The 1 Unified Analytics Engine for Big Data. URL https://databricks.com/spark/about. Accessed on Apr. 18, 2018.

[11] Dstat. URL http://dag.wiee.rs/home-made/dstat/.

[12] Real time fraud detection through streaming analytics. URL https://blog.sitm.ac.in/fraud-detection-streaming-analytics/. Accessed on Dec. 17, 2017.

[13] Data management platform, solutions and big data analysis. URL https://hortonworks.com/. Accessed on Dec. 16, 2017.

[14] Explore spark sql and its performance using tpc-ds workload. URL https://developer.ibm.com/patterns/explore-spark-sql-and-its-performance-using-tpc-ds-workload/. Accessed on Apr. 19, 2018.

[15] Hadoop and mapreduce. URL https://www.packtpub.com/books/content/hadoop-and-mapreduce. Accessed on Dec. 05, 2017.

[16] Apache parquet. URL https://parquet.apache.org/.

[17] Snappy. URL https://google.github.io/snappy/.

[18] Ted cruz using firm that harvested data on millions of unwitting facebook users. URL https://www.theguardian.com/us-news/2015/dec/11/senator-ted-cruz-president-campaign-facebook-user-data. Accessed on Jul. 19, 2018.

[19] Gdpr: Getting ready for the new eu general data protection regulation, May 2016. URL https://www.infolawgroup.com/2016/05/articles/gdpr/gdpr-getting-ready-for-the-new-eu-general-data-protection-regulation/. Accessed on Dec. 14, 2017.

[20] Iso - international organization for standardization, Nov 2016. URL https://www.iso.org/standard/50375.html. Accessed on Dec. 17, 2017.

[21] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. Order preserving encryption for numeric data. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 563–574, New York, NY, USA, 2004. ACM. ISBN 1-58113-859-8. doi: 10.1145/1007568.1007632. URL http://doi.acm.org/10.1145/1007568.1007632.

[22] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1383–1394. ACM, 2015.

[23] John Black and Phillip Rogaway. Ciphers with arbitrary finite domains. In *Cryptographers' Track at the RSA Conference*, pages 114–130. Springer, 2002.

[24] How-Shen Chang. International data encryption algorithm. *jmu. edu, googleusercontent. com, Fall*, 2004.

[25] Jean-Sébastien Coron, David Naccache, and Mehdi Tibouchi. *Public Key Compression and Modulus Switching for Fully Homomorphic Encryption over the Integers*, pages 446–464. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-29011-4. doi: 10.1007/978-3-642-29011-4_27. URL https://doi.org/10.1007/978-3-642-29011-4_27.

[26] Victor Costan and Srinivas Devadas. Intel sgx explained.

[27] Nik Cubrilovic. Rockyou hack: From bad to worse, Dec 2009. URL https://techcrunch.com/2009/12/14/rockyou-hack-security-myspace-facebook-passwords/. Accessed on Oct. 25, 2017.

[28] Joan Daemen and Vincent Rijmen. Rijndael/aes. In *Encyclopedia of Cryptography and Security*, pages 520–524. Springer, 2005.

[29] R Davis. The data encryption standard in perspective. *IEEE Communications Society Magazine*, 16(6):5–9, 1978.

[30] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[31] F Betül Durak and Serge Vaudenay. Breaking the ff3 format-preserving encryption standard over small domains. In *Annual International Cryptology Conference*, pages 679–707. Springer, 2017.

[32] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing*, STOC '09, pages 169–178, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-506-2. doi: 10.1145/1536414.1536440. URL http://doi.acm.org/10.1145/1536414.1536440.

[33] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. *The Google file system*, volume 37. ACM, 2003.

[34] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *Journal of computer and system sciences*, 28(2):270–299, 1984.

[35] Shay Gueron, Adam Langley, and Yehuda Lindell. Aes-gcm-siv: Specification and analysis. 2017.

[36] Mark Hachman. The price of free: how apple, facebook, microsoft and google sell you to advertisers, Oct 2015. URL https://www.pcworld.com/article/2986988/privacy/the-price-of-free-how-apple-facebook-microsoft-and-google-sell-you-to-advertisers.html. Accessed on Dec. 12, 2017.

[37] hortonworks spark. hortonworks-spark/shc, Dec 2017. URL https://github.com/hortonworks-spark/shc. Accessed on Dec. 16, 2017.

[38] John M. (Intel). Aes-gcm encryption performance on intel® xeon® e5 v3 processors, Jun 2017. URL https://software.intel.com/en-us/articles/aes-gcm-encryption-performance-on-intel-xeon-e5-v3-processors.

[39] Ricardo Macedo, Joao Paulo, Rogerio Pontes, Bernardo Portela, Tiago Oliveira, Miguel Matos, and Rui Oliveira. A practical framework for privacy-preserving nosql databases. *2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS)*, 2017. doi: 10.1109/srds.2017.10.

[40] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research*, 17(1):1235–1241, 2016.

[41] Raghunath Othayoth Nambiar and Meikel Poess. The making of tpc-ds. In *Proceedings of the 32Nd International Conference on Very Large Data Bases*, VLDB '06, pages 1049–1058. VLDB Endowment, 2006. URL http://dl.acm.org/citation.cfm?id=1182635.1164217.

[42] Muhammad Naveed, Seny Kamara, and Charles V Wright. Inference attacks on property-preserving encrypted databases. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 644–655. ACM, 2015.

[43] Salman Niazi, Mahmoud Ismail, Seif Haridi, Jim Dowling, Steffen Grohsschmiedt, and Mikael Ronström. Hopsfs: Scaling hierarchical file system metadata using newsql databases. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 89–104, Santa Clara, CA, 2017. USENIX Association. ISBN 978-1-931971-36-2. URL https://www.usenix.org/conference/fast17/technical-sessions/presentation/niazi.

[44] Hilary Osborne and Hannah Jane Parkinson. Cambridge analytica scandal: the biggest revelations so far, Mar 2018. URL https://www.theguardian.com/uk-news/2018/mar/22/cambridge-analytica-scandal-the-biggest-revelations-so-far. Accessed on Jul. 18, 2018.

[45] Pascal Paillier et al. Public-key cryptosystems based on composite degree residuosity classes. In *Eurocrypt*, volume 99, pages 223–238. Springer, 1999.

[46] Antonis Papadimitriou, Ranjita Bhagwan, Nishanth Chandran, Ramachandran Ramjee, Andreas Haeberlen, Harmeet Singh, Abhishek Modi, and Saikrishna Badrinarayanan. Big data analytics over encrypted datasets with seabed. In *OSDI*, pages 587–602, 2016.

[47] Meikel Poess, Raghunath Othayoth Nambiar, and David Walrath. Why you should run tpc-ds: A workload analysis. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 1138–1149. VLDB Endowment, 2007. ISBN 978-1-59593-649-3. URL http://dl.acm.org/citation.cfm?id=1325851.1325979.

[48] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. Cryptdb. *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles - SOSP 11*, 2011. doi: 10.1145/2043556.2043566.

[49] Raluca Ada Popa, Frank H Li, and Nickolai Zeldovich. An ideal-security protocol for order-preserving encoding. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 463–477. IEEE, 2013.

[50] Do Le Quoc, Martin Beck, Pramod Bhatotia, Ruichuan Chen, Christof Fetzer, and Thorsten Strufe. Privapprox: Privacy-preserving stream analytics. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 659–672, Santa Clara, CA, 2017. USENIX Association. ISBN 978-1-931971-38-6. URL https://www.usenix.org/conference/atc17/technical-sessions/presentation/quoc.

[51] Ronald L Rivest. The rc5 encryption algorithm. In *International Workshop on Fast Software Encryption*, pages 86–96. Springer, 1994.

[52] Seref Sagiroglu and Duygu Sinanc. Big data: A review. URL http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6567202&tag=1.

[53] Bruce Schneier. Description of a new variable-length key, 64-bit block cipher (blowfish). In *International Workshop on Fast Software Encryption*, pages 191–204. Springer, 1993.

[54] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*, pages 1–10. IEEE, 2010.

[55] Dawn Xiaoding Song, David Wagner, and Adrian Perrig. Practical techniques for searches on encrypted data. In *Security and Privacy, 2000. S&P 2000. Proceedings. 2000 IEEE Symposium on*, pages 44–55. IEEE, 2000.

[56] Transaction Processing Performance Council (TPC). Tpc benchmark ds. 2018.

[57] Stephen Tu, M Frans Kaashoek, Samuel Madden, and Nickolai Zeldovich. Processing analytical queries over encrypted data. In *Proceedings of the VLDB Endowment*, volume 6, pages 289–300. VLDB Endowment, 2013.

[58] Reynold S Xin, Joseph E Gonzalez, Michael J Franklin, and Ion Stoica. Graphx: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems*, page 2. ACM, 2013.

[59] Reynold S Xin, Josh Rosen, Matei Zaharia, Michael J Franklin, Scott Shenker, and Ion Stoica. Shark: Sql and rich analytics at scale. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of data*, pages 13–24. ACM, 2013.

[60] Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, and Ion Stoica. Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters. *HotCloud*, 12:10–10, 2012.

[61] Wenting Zheng, Ankur Dave, Jethro G Beekman, Raluca Ada Popa, Joseph E Gonzalez, and Ion Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *NSDI*, pages 283–298, 2017.

A

DETALHES DOS RESULTADOS

Interrogação 24	Spark					HBase				
	#1	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**	**#9**	**#10**
CPU(%)	86.72	7.45	5.00	3.71	7.32	25.25	0.09	2.86	15.11	0.22
Memória (GB)	2.74	3.07	2.45	1.84	3.14	1.47	3.05	3.09	3.06	3.04
Disco - Leitura (KB/s)	≈ 0	0.03	0.10	≈ 0	0.04	0.01	≈ 0	718.79	44 376	≈ 0
Disco - Escrita (KB/s)	49	610.79	428.18	277.52	577.43	30.91	6.57	7.09	6.66	6.38
Rede - Receção (MB/s)	0.02	3.61	2.54	1.80	3.6	0.01	≈ 0	0.01	0.07	≈ 0
Rede - Envio (MB/s)	0.02	0.26	0.23	0.08	0.19	≈ 0	≈ 0	0.69	10.17	0.02

Tabela 8: Resultados do *dstat* para a interrogação 24

Interrogação 27	Spark					HBase				
	#1	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**	**#9**	**#10**
CPU(%)	86.69	3.75	7.62	3.71	10.08	25.26	0.11	0.20	16.48	0.12
Memória (GB)	2.81	2.81	2.45	1.78	2.77	25.26	3.09	3.09	3.05	3.04
Disco - Leitura (KB/s)	≈ 0	≈ 0	≈ 0	≈ 0	≈ 0	≈ 0	≈ 0	≈ 0	44 698	≈ 0
Disco - Escrita (KB/s)	45.13	297.24	618.53	303.51	662.39	31.05	6.68	6.53	6.98	6.43
Rede - Receção (MB/s)	0.02	2.47	5.12	2.47	6.24	0.011	≈ 0	≈ 0	0.072	≈ 0
Rede - Envio (MB/s)	0.01	0.16	0.33	0.16	0.04	≈ 0	≈ 0	≈ 0	15.69	0.02

Tabela 9: Resultados do *dstat* para a interrogação 27

Interrogação 31	Spark					HBase				
	#1	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**	**#9**	**#10**
CPU(%)	87.44	6.95	8.92	3.93	7.18	25.24	0.15	0.15	14.04	6.17
Memória (GB)	2.60	2.59	2.82	2.12	2.78	1.49	3.03	3.04	3.05	3.02
Disco - Leitura (KB/s)	≈ 0	≈ 0	≈ 0	≈ 0	≈ 0	≈ 0	≈ 0	≈ 0	41 762	15 906
Disco - Escrita (KB/s)	53.67	153.21	405.61	151.67	275.07	32.72	6.48	6.48	6.85	6.42
Rede - Receção (MB/s)	0.05	1.41	3.46	1.33	2.31	0.01	≈ 0	≈ 0	≈ 0	≈ 0
Rede - Envio (MB/s)	0.04	0.2	0.46	0.19	0.28	≈ 0	≈ 0	0.04	5.91	1.54

Tabela 10: Resultados do *dstat* para a interrogação 31

Interrogação 73	Spark					HBase				
	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10
CPU(%)	87.41	1.96	4.2	4.21	7.18	25.25	0.12	0.47	14.65	0.19
Memória (GB)	2.51	2.36	2.01	1.77	2.81	1.47	3.05	3.09	3.56	3.04
Disco - Leitura (KB/s)	≈ 0	≈ 0	≈ 0	≈ 0	≈ 0	≈ 0	≈ 0	≈ 0	41 879	≈ 0
Disco - Escrita (KB/s)	0.05	0.03	0.04	0.04	0.12	0.03	≈ 0	≈ 0	≈ 0	0.01
Rede - Receção (MB/s)	≈ 0	≈ 0	1.61	1.83	6.16	≈ 0	≈ 0	≈ 0	≈ 0	≈ 0
Rede - Envio (MB/s)	0.01	0.02	0.03	0.03	0.06	≈ 0	≈ 0	0.02	9.55	0.02

Tabela 11: Resultados do *dstat* para a interrogação 73

Interrogação 40	Spark					HBase				
	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10
CPU(%)	86.69	6.72	6.69	4.22	4.68	25.13	0.14	0.83	13.26	3.53
Memória (GB)	2.79	2.72	2.86	2.26	3.01	1.49	3.01	3.03	3.04	3.001
Disco - Leitura (KB/s)	≈ 0	≈ 0	≈ 0	0.15	≈ 0	0.06	≈ 0	0.55	42 114	8 858
Disco - Escrita (KB/s)	0.05	0.29	0.39	0.18	0.19	0.03	0.08	0.01	0.01	0.01
Rede - Receção (MB/s)	0.02	2.2	2.72	≈ 0	1.59	0.01	≈ 0	≈ 0	0.05	0.09
Rede - Envio (MB/s)	0.01	0.24	0.35	0.14	0.18	≈ 0	≈ 0	0.21	6.16	0.86

Tabela 12: Resultados do *dstat* para a interrogação 40

Interrogação 81	Spark					HBase				
	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10
CPU(%)	86.40	2.31	37.49	10.69	19.26	25.27	0.33	28.42	0.09	0.21
Memória (GB)	2.81	3.26	3.09	2.60	3.79	1.50	3.03	3.04	3.05	3.02
Disco - Leitura (KB/s)	≈ 0	≈ 0	0.44	≈ 0	≈ 0	≈ 0	≈ 0	9 330	≈ 0	≈ 0
Disco - Escrita (KB/s)	0.04	0.02	0.59	0.35	0.76	0.03	0.07	0.03	0.01	0.01
Rede - Receção (MB/s)	0.32	0.02	6.03	2.93	5.99	0.01	≈ 0	0.09	≈ 0	≈ 0
Rede - Envio (MB/s)	0.24	0.02	0.23	0.25	0.64	≈ 0	≈ 0	14.19	0.24	0.86

Tabela 13: Resultados do *dstat* para a interrogação 81

Interrogação 70	Spark					HBase				
	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10
CPU(%)	86.48	7.67	4.42	3.45	7.82	25.33	0.11	0.09	23.75	0.11
Memória (GB)	2.81	2.88	2.47	2.75	3.35	1.50	3.03	3.04	3.06	3.02
Disco - Leitura (KB/s)	≈ 0	≈ 0	≈ 0	≈ 0	≈ 0	0.02	≈ 0	≈ 0	44 592	≈ 0
Disco - Escrita (KB/s)	0.05	0.46	0.26	0.26	0.65	0.03	0.01	0.01	0.01	0.01
Rede - Receção (MB/s)	0.03	4.15	2.36	2.09	4.18	0.01	≈ 0	≈ 0	0.12	≈ 0
Rede - Envio (MB/s)	0.02	0.17	0.16	0.14	0.17	≈ 0	0.03	≈ 0	12.28	≈ 0

Tabela 14: Resultados do *dstat* para a interrogação 70

Interrogação 82	Spark					HBase				
	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10
CPU(%)	86.59	8.09	5.02	3.70	3.29	25.26	21.85	0.09	9.29	0.11
Memória (GB)	2.69	2.42	1.93	1.95	3.24	1.49	3.06	3.09	3.05	3.04
Disco - Leitura (KB/s)	≈ 0	0.02	≈ 0	≈ 0	≈ 0	≈ 0	22 744	≈ 0	≈ 0	≈ 0
Disco - Escrita (KB/s)	0.05	0.28	0.17	0.09	0.12	0.03	≈ 0	0.01	0.01	0.01
Rede - Receção (MB/s)	0.05	4.14	2.18	1.37	1.66	0.03	0.08	≈ 0	0.04	≈ 0
Rede - Envio (MB/s)	0.01	0.19	0.12	0.02	0.08	≈ 0	7.71	≈ 0	1.37	≈ 0

Tabela 15: Resultados do *dstat* para a interrogação 82

Tabela	Coluna	Segurança
call_center	key	PLT
	cc_call_center_sk	DET
	cc_call_center_id	DET
	cc_closed_date_sk	DET
	cc_open_date_sk	DET
	default	STD
catalog_page	key	PLT
	cp_catalog_page_sk	DET
	cp_catalog_page_id	DET
	cp_start_date_sk	DET
	cp_end_date_sk	DET
	default	STD
catalog_sales	key	PLT
	cs_sold_date_sk	DET
	cs_sold_time_sk	DET
	cs_ship_date_sk	DET
	cs_bill_customer_sk	DET
	cs_bill_cdemo_sk	DET
	cs_bill_hdemo_sk	DET
	cs_bill_addr_sk	DET
	cs_ship_customer_sk	DET
	cs_ship_cdemo_sk	DET
	cs_ship_hdemo_sk	DET
	cs_ship_addr_sk	DET
	cs_call_center_sk	DET
	cs_catalog_page_sk	DET
	cs_ship_mode_sk	DET
	cs_warehouse_sk	DET
	cs item sk	DET
	cs promo sk	DET
	default	STD

Tabela	Coluna	Segurança
catalog_returns	key	PLT
	cr_returned_date_sk	DET
	cr_returned_time_sk	DET
	cr_item_sk	DET
	cr_refunded_customer_sk	DET
	cr_refunded_cdemo_sk	DET
	cr_refunded_hdemo_sk	DET
	cr_refunded_addr_sk	DET
	cr_returning_customer_sk	DET
	cr_returning_cdemo_sk	DET
	cr_returning_hdemo_sk	DET
	cr_returning_addr_sk	DET
	cr_call_center_sk	DET
	cr_catalog_page_sk	DET
	cr_ship_mode_sk	DET
	cr_warehouse_sk	DET
	cr_reason_sk	DET
	default	STD
date_dim	key	PLT
	d_date_sk	DET
	d_date_id	DET
	d_date	OPE
	d_month_seq	OPE
	d_year	DET
	d_dom	OPE
	d_qoy	DET
	default	STD
income_band	key	PLT
	ib_income_band_sk	DET
	default	STD

Tabela 16: Esquema da base de dados(parte 1)

Tabela	Coluna	Segurança
customer	key	PLT
	c_customer_sk	DET
	c_customer_id	DET
	c_current_cdemo_sk	DET
	c_current_hdemo_sk	DET
	c_current_addr_sk	DET
	c_first_shipto_date_sk	DET
	c_first_sale_date_sk	DET
	c_birth_country	DET
	default	STD
customer_address	key	PLT
	ca_address_sk	DET
	ca_address_id	DET
	ca_state	DET
	ca_zip	DET
	ca_country	DET
	default	STD
customer_demographics	key	PLT
	cd_demo_sk	DET
	cd_gender	DET
	cd_marital_status	DET
	cd_educational_status	DET
	default	STD
household_demographics	key	PLT
	hd_demo_sk	DET
	hd_income_band_sk	DET
	hd_buy_potential	OPE
	hd_vehicle_count	OPE
	cs item sk	STD

Tabela	Coluna	Segurança
inventory	key	PLT
	inv_date_sk	DET
	inv_item_sk	DET
	inv_warehouse_sk	DET
	inv_quantity_on_hand	OPE
	default	STD
item	key	PLT
	i_item_sk	DET
	i_item_id	DET
	i_item_current_price	OPE
	i_color	DET
	i_manufact	DET
	default	STD
promotion	key	PLT
	p_promo_sk	DET
	p_promo_id	DET
	p_start_date_sk	OPE
	p_end_date_sk	DET
	p_item_sk	DET
	default	STD
reason	key	DET
	r_reason_sk	DET
	r_reason_id	DET
	default	STD
ship_mode	key	PLT
	sm_ship_mode_sk	DET
	sm_ship_mode_id	DET
	default	STD

Tabela 17: Esquema da base de dados(parte 2)

Tabela	Coluna	Segurança
store	key	PLT
	s_store_sk	DET
	s_store_id	DET
	s_closed_date_sk	DET
	s_county	DET
	s_state	DET
	s_zip	DET
	s_market_zip	DET
	default	STD
store_returns	key	PLT
	sr_returned_date_sk	DET
	sr_returned_time_sk	DET
	sr_item_sk	DET
	sr_customer_sk	DET
	sr_cdemo_sk	DET
	sr_hdemo_sk	DET
	sr_addr_sk	DET
	sr_store_sk	DET
	sr_reason_sk	DET
	sr_ticket_number_sk	DET
	default	STD
store_sales	key	PLT
	ss_sold_date_sk	DET
	ss_sold_time_sk	DET
	ss_item_sk	DET
	ss_customer_sk	DET
	ss_cdemo_sk	DET
	ss_hdemo_sk	DET
	ss_addr_sk	DET
	ss_store_sk	DET
	ss_promo_sk	DET
	ss_ticket_number	DET
	default	STD
time_dim	key	PLT
	t_time_sk	DET
	t_time_id	DET
	default	STD
warehouse	key	PLT
	w_warehouse_sk	DET
	w_warehouse_id	DET
	default	STD

Tabela	Coluna	Segurança
web_page	key	PLT
	wp_web_page_sk	DET
	wp_web_page_id	DET
	wp_rec_start_date	DET
	wp_rec_end_date	DET
	wp_creation_date_sk	DET
	wp_access_date_sk	DET
	wp_customer_sk	DET
	default	STD
web_returns	key	PLT
	wr_returned_date_sk	DET
	wr_returned_time_sk	DET
	wr_item_sk	DET
	wr_refunded_customer_sk	DET
	wr_refunded_cdemo_sk	DET
	wr_refunded_hdemo_sk	DET
	wr_refunded_addr_sk	DET
	wr_returning_cdemo_sk	DET
	wr_returning_hdemo_sk	DET
	wr_returning_addr_sk	DET
	wr_web_page_sk	DET
	default	STD
web_site	key	PLT
	ws_sold_date_sk	DET
	ws_sold_time_sk	DET
	ws_ship_date_sk	DET
	ws_item_sk	DET
	ws_bill_customer_sk	DET
	ws_bill_cdemo_sk	DET
	ws_bill_hdemo	DET
	ws_bill_addr_sk	DET
	ws_ship_customer_sk	DET
	ws_ship_cdemo_sk	DET
	ws_ship_hdemo_sk	DET
	ws_ship_addr	DET
	ws_web_page_sk	DET
	ws_web_site_sk	DET
	ws_ship_mode_sk	DET
	ws_warehouse_sk	DET
	ws_promo_sk	DET
	default	STD

Tabela 18: Esquema da base de dados(parte 3)